A Functional-Level Simulator for the Configurable (Many-Core) PRAM-Like REPLICA Architecture

by

Andreas Lööw

LIU-IDA/LITH-EX-A--12/033--SE

2012-06-25
Final Thesis

A Functional-Level Simulator for the Configurable (Many-Core) PRAM-Like REPLICA Architecture

by

Andreas Lööw

LIU-IDA/LITH-EX-A--12/033--SE

2012-06-25

Supervisor: Erik Hansson
Examiner: Christoph Kessler
Abstract

This master’s thesis discusses the design and implementation of a simulator for the REPLICA architecture, a many-core PRAM-like machine. REPLICA provides a programming model that seemingly cannot be provided by mainstream hardware without significant slowdown compared to traditional models. This also implies that it is difficult to simulate REPLICA’s programming model on mainstream hardware. Simulator design decisions are described and the resulting simulator is evaluated and compared to existing simulators, where we see that the simulator presented in this thesis is the fastest of them. As seen from the discussion focus in the thesis, most efforts were directed towards simulator execution speed rather than user-facing features.

Acknowledgments

Thanks to Martti Forsell, for help with the REPLICA architecture and comments on the report, Erik Hansson, for acting as supervisor, and Christoph Kessler, for acting as examiner.

License

This work is licensed under a Creative Commons Attribution-NonCommercial 3.0 Unported License[^1], except figures 3.2, 3.3 and 3.4 which are licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported License[^2].

[^1]: http://creativecommons.org/licenses/by-nc/3.0/
[^2]: http://creativecommons.org/licenses/by-nc-sa/3.0/
# Contents

1 Introduction ............................................. 6
   1.1 Problem formulation and motivation ................ 6
   1.2 Notational conventions ............................. 7
       1.2.1 Reference handling ............................ 7
   1.3 Simulator terminology and background .......... 7
       1.3.1 Terminology .................................. 8
       1.3.2 Terminology used in this thesis: observability and modelling accuracy ... 9
   1.4 Some requirement’s implementation implications 11
   1.5 Thesis outline .................................... 12
   1.6 Summary and conclusion ............................ 12

2 Requirements ............................................ 13
   2.1 Core requirements and features ................... 13
   2.2 Extra requirements and features ................... 14

3 The PRAM Model and the REPLICA Architecture .... 15
   3.1 The Parallel Random Access Machine (PRAM) ...... 15
   3.2 The REPLICA architecture ......................... 17
       3.2.1 A crash-course in REPLICA assembly programming ... 18
       3.2.2 Some REPLICA implementation details ............ 21
   3.3 Conclusion ........................................ 26

4 Survey of Simulation Methods ......................... 27
   4.1 Direct interpretation ............................... 28
       4.1.1 A basic interpreter for a simple machine .... 29
       4.1.2 Problems with direct interpretation ......... 32
       4.1.3 Debugging ................................... 35
       4.1.4 Summary and conclusion ....................... 35
   4.2 Static translation ................................ 36
       4.2.1 Summary and conclusion ........................ 36
   4.3 Dynamic translation ................................ 37
       4.3.1 What to translate ................................ 37
       4.3.2 When to translate ................................ 37
       4.3.3 Simple just-in-time compilation ............... 38
       4.3.4 Advanced just-in-time compilation ............. 39
       4.3.5 Practical aspects: code generation and JIT frameworks 39
       4.3.6 Related work .................................. 40
       4.3.7 Debugging ..................................... 41
       4.3.8 Summary and conclusion ......................... 41
   4.4 Short outline of other simulation methods ....... 41
Chapter 1

Introduction

Accurate and reliable machine simulation helps comprehensive design space exploration (DSE) within time-constrained projects and hardware/software co-design of microprocessor architectures. A flexible (research) machine simulator should fulfill a large set of criteria, amongst other: high speed simulation, accurate modelling, instrumentation and state observability (Jones 2010).

Informally, but adequately precise and clear for this thesis, machine simulation can be defined as follows (Keller, Kessler, and Träff 2001; Harris 1994).

Definition 1. A simulation of machine $M_1$ on machine $M_2$ is an algorithm that allows any instruction from $M_1$ to be executed on $M_2$. We refer to machine $M_1$ as the guest machine, and to machine $M_2$ as the host machine.

This “host and guest” convention will be naturally extended as, and when, needed. For example, “host instructions” may be used to refer to some set of instructions on the host machine. The definition, of course, is of informal nature because we have not in any satisfactory way specified what we mean by a “machine”, an “algorithm” or even an “instruction”. This, however, as also stated before, is not important here. Another perspective on machine simulation is given in the context of virtualization – trying to relate, amongst others, simulation and virtualization is the main goal of (the soon to follow) section 1.3. Virtualization can be seen as the construction of an isomorphism that maps a virtual guest system to a host system (Smith and Nair 2005).

In this thesis a special type of machine is to be simulated, namely, a parallel machine. That is, a machine with two or more processors. Throughout this thesis, the terms “processor” and “core” will be considered synonyms, as we (here) never need to differentiate between the situation where two cores are located on the same processor or located on two distinct processors.

1.1 Problem formulation and motivation

The removing performance and programmability limitations of chip multiprocessor architecture (REPLICA) project aims at developing a configurable emulated shared memory machine (CESM) architecture realizing the parallel random access machine (PRAM) model of parallel computation. The main topic of this thesis is the design and implementation of a simulator for the REPLICA architecture. Sometimes, we will simply refer to “REPLICA” instead of the longer, “REPLICA architecture”, or “REPLICA machine”.

The simulator will read REPLICA assembly source code and simulate the execution of the program the source code represents as if it would run on a physical REPLICA machine.

http://vtt.fi/sites/replica

6
There already exists a (non-publicly available) simulator for REPLICA’s predecessor, the TOTAL ECLIPSE architecture (Forsell 2010). This simulator is however very difficult to maintain as the whole program consists of approximately 46000 lines Pascal code – all in one file. In addition to this, the program exhibits a large amount of bugs, making it difficult to use because of frequent crashes. The simulator is, moreover, for many purposes too slow and only runs on Mac OS (using the Mac OS Carbon framework).

˚Akesson 2012 used the old simulator to build a compiler back-end for REPLICA. The thesis gives some further details, and screenshots, of the old simulator. Along these lines, a manual for an older version of the simulator is available and provides some more details, see Forsell 1997.

The simulator designed in this thesis aims at solving these problems. It will therefore be, to as a large extent as possible, platform-independent (by delegating platform-dependent operations to multi-platform libraries) and adhere to acknowledged software-engineering principles. Furthermore, the aim is to develop a faster simulator than the currently existing, possibly by utilizing multiple processors on the host computer if such an approach turns out to be beneficial.

### 1.2 Notational conventions

Shorter code fragments are displayed inline and formatted like this. To describe short programs and algorithms a C++-like language is used. Not all functions and types (e.g., `structs` and `classes`) are defined in detail. The existence of some (useful) set of functions are sometimes simply assumed and only described informally in English; in most cases because the implementation details of the functions involved are unimportant and uninteresting. Consider for example the following short program, where `get_nth_happy_number(int n)` is a function returning the $n$th happy number.

```c
int main() {
    for (int i = 1; i <= 10; ++i)
        print("Another happy number: ",
              + get_nth_happy_number(i) + "\n");
}
```

Listing 1.1: A short example program, that will print the ten first happy numbers.

Even though we have not provided the implementation details of `get_nth_happy_number` it should still be clear what the program does. The reader should also note that some amount of pseudocode will be used, as above where `+` denotes string concatenation, integer-to-string conversion happens automatically, and `print` (a function not included in the C++ standard library) will display its argument to the user running the program by e.g. printing it to `stdout`.

### 1.2.1 Reference handling

References are handled as usual, but the references that are placed in footnotes should only be considered “further readings” and not actual references used to support factual claims.

### 1.3 Simulator terminology and background

There are numerous terms used in relation to computer simulation. Among other: simulators, emulators, abstract machines, system virtual machines (VMs), and application/process virtual machines. The meaning of these terms

---

2 A positive integer $n_0$ is a happy number iff there exists an $i \in \mathbb{N}$ such that $n_i = 1$ in the sequence $(n_0, n_1, n_2, \ldots)$ where $n_j$ is the sum of the squares of the digits of $n_{j-1}$, for each $j \in \mathbb{N}$. See e.g. [http://mathworld.wolfram.com/HappyNumber.html](http://mathworld.wolfram.com/HappyNumber.html)
varies (widely) between authors. This section (tries to) give an overview of the terms’ different uses, and alternative terms (where applicable). We will conclude that the aforementioned terms are too vague to be of any use here, and therefore introduce and discuss the (to the author’s knowledge) less commonly used terminology that is used in this thesis, and at the same time provide some background on computer simulation.

1.3.1 Terminology

To begin with, the terms computer and machine are often used interchangeably (in this thesis as well). Some conventions do exist however, e.g. one rarely speaks of “virtual computers” (rather than “virtual machines”) – but even if one would do, the meaning would still be the same.

From the author’s limited experience emulation (often) refers to when only directly observable aspects (external aspects) of the machine are to be modelled, and simulation (often) refers to when also non-directly observable aspects (internal aspects) of the computer are to be modelled. The terms architectural-level simulation and functional-level simulation are also used sometimes, and refer to simulators and emulators (as defined in the previous sentence), respectively. Instruction-level simulation is sometimes used as a synonym for functional-level simulation.

The term virtual machine is often used to refer to programs such as VMware’s products, e.g. VMware Workstation. That is, system virtual machines. A system virtual machine is a “whole” computer, running a full (sometimes modified, sometimes unmodified) operating system for real, physical, computers. There is however another type of virtual machines as well; namely, process virtual machines, also called application virtual machines. Two well-known examples being the family of Java virtual machines (JVMs) and Microsoft’s common language runtime (CLR). In contrast to system virtual machines, process virtual machines do not run a full operating system. Rather, as implied by the name, only one process at a time is running on the virtual machine (or, more generally, one application – if the application could be said to consists of multiple processes).

Lastly, the term abstract machine seems less common in simulation contexts and more commonly used in the context of e.g. the theory of computation, compiler construction and semantics of programming languages; where it is, e.g., used for referring to models of computation such as Turing machines. Abstract machines in this sense do not, directly, play an important role in this thesis.

As previously mentioned, other authors use other terminology or assign different meanings to the terms discussed above. For example, Liu p. 339, categorizes assembly language simulators to be either an instruction set simulator (ISS) or a processor (computer) architecture simulator, roughly corresponding to, what was previously called here, emulator and simulator, respectively. Other authors assign other meanings to the terms defined and discussed above. Some, for example, consider the terms simulator and emulator to be synonyms. Others, use emulation to mean (partly or fully) hardware-supported simulation, where “simulation” in this context then refers to both simulation and emulation as defined above (i.e., whether to model internal aspects or not, respectively).

To develop a uniform framework to describe, relate and reason about the above notions are, however, not the goal of (or even essential for) this thesis. Because of this, the above (vague) outline will not be refined further. The important part is the conclusion that different authors use different terminology.

---

3In this section there are a few “others, . . .” and “some authors, . . .” instead of proper references. Finding such “proper references” for all terms would be rather time consuming and is beyond the scope of this thesis. Furthermore, inferring from usage alone what some author means by some term might not always be an entirely straightforward process – so the discussion in this section should be considered a summary of the terminology usage the author was subject to during the literature review for this thesis (which might or might not reflect mainstream usage, and can as such therefore not be treated as statements of facts).

4E.g., Oracle’s HotSpot/OpenJDK [http://openjdk.java.net], Jikes RVM [http://jikesrvm.org]. For further details see Lindholm and Yellin [1999].

5Microsoft’s implementation of the common language infrastructure (CLI), see e.g. [http://msdn.microsoft.com/en-us/netframework/aa569283] for more information.
1.3 Simulator terminology and background

Because of this terminological mess no special meaning will be assigned to the different terms in this thesis. Instead, the terms “simulator” and “virtual machine” will be used synonymously and refer to all of the above notions, rather than the awkward “simulator/emulator” term or something similar. This convention holds true retroactively for the introduction of this thesis as well. “Simulator” (in this thesis, and consequently also “virtual machine”) is therefore simply defined as follows.

**Definition 2.** A simulator is the implementation of the algorithm part from the machine simulation definition (definition[1]), i.e. a program that “drives” a simulation.

As previously stated, we consider the above terminology insufficient to describe simulators in more detail, the following section will therefore introduce the terminology used in this thesis.

1.3.2 Terminology used in this thesis: observability and modelling accuracy

The (important) realization that whether a simulator models “internal aspects” of the machine is not a simple yes/no answer, and consequently that the simulator/emulator dichotomy therefore is not sufficient to describe a simulator’s functionality in any depth, directly leads us to the following question: what should we use instead? The answer to this question proposed in this thesis is adopted from Pees[2003]. Put simply, we will use the notions of “spatial accuracy” and “temporal accuracy” instead of categorizing a simulator as a simulator or emulator (where the first usage of the term “simulator” refers to how it is used in this thesis, and the second usage of the term refers to when it is used to mean that internal aspects are modelled).

Not overly atypical when modeling any kind of phenomena is the tradeoff between modeling accuracy and implementation cost; the latter with respect to time, and consequently, also often with respect to monetary cost. We refer to the level of accuracy/detail as different levels of abstractions or layers of abstractions. A simulator implemented in terms of a high level of abstraction does not expose low-level details to the user, but such implementations in turn probably carry a (relatively) low implementation cost. In contrast, a simulator implemented in terms of a low-level abstraction layer provides the user with a more detailed view of the system/phenomena simulated, but also carries a higher implementation cost than the corresponding simulator implemented in high-level terms. Furthermore, often, because high-level simulators can ignore low-level aspects they can execute the simulation faster than the corresponding low-level simulator.

Obviously, there are multiple levels of abstraction – not just two levels called “high” and “low”. For example, a simulation can be carried out on (in ascending level of abstractions) transistor level, gate level, register transfer level (RTL) or instruction set architecture level (to name a few). As seen in these examples, machine modeling abstraction layers are mainly characterized by the level of detail of the real hardware that are represented in the layer. That is, the abstraction layers are identified by the smallest hardware structures that are the atomic elements in the model – i.e., elements without internal structure that cannot be divided further (in the model), e.g. transistors or logic gates in transistor-level or gate-level layers, respectively.

To further clarify, there are two important aspects that can be varied between layers. Both the spatial accuracy and temporal accuracy. Spatial accuracy refers to which elements that are the atomic elements in the layer (as described in the previous paragraph), and temporal accuracy refers to how frequent machine state will be made visible to the user (i.e., the resolution of time). Three examples of temporal accuracy levels are instruction accurate, cycle accurate or quasi-continuously (the two former being of discrete nature). In instruction-accurate simulation machine state is visible to the user between instructions, analogously for cycle-accurate simulation, and lastly quasi-continuous simulation means that machine state can be reproduced at any time. If one considers a debugger for any mainstream language, let us say C, temporal accuracy would here correspond to the smallest possible step that is possible to do when stepping through the program, for example source code lines or assembly instructions.

---

The reason “functional-level” was used in the title of this work is that using somewhat non-standard terminology there seemed like a bad idea.
For any concrete simulator these two aspects (spatial and temporal) are fixed, e.g. a simulator can have instruction set level spatial accuracy combined with instruction-, cycle- or phase-accurate temporal accuracy. However, temporal and spatial accuracy cannot, in general, be varied independently of each other. Because, for example, increasing the resolution of time, more and more machine architecture aspects must be made visible to the user (or at least modelled internal) as these parts constitute the parts of the architecture causing the specific operation timing. Often, the two notions of accuracy used in a simulator coincide, and we can then say just (e.g.) “cycle accurate simulation” instead of (the verbose alternative) “spatial cycle accurate and temporal cycle accurate simulation”.

There are some important aspects to take note of here. Consider some element \( A \) from some abstraction level receiving some input \( \hat{X} \) and that gives some output \( \hat{Y} \). Furthermore, let \( \hat{Y}(t) \) and \( \hat{X}(t) \) denote the value of \( \hat{Y} \) and \( \hat{X} \) at time instance \( t \), respectively. If for any time instance \( t_0 \), \( \hat{Y}(t_0) \) is independent (does not depend on) of \( \hat{X}(t_1) \) for any other time instance \( t_1 \neq t_0 \), then \( A \) can be modelled as a function (in the mathematical sense), i.e. without any memory of previous inputs (internal state). This is illustrated in figure 1.1a. (What kind of input \( A \) receives depend on the surrounding elements, and thus the level of abstraction \( A \) belongs to.)

If \( \hat{Y}(t_0) \) is not independent of all \( \hat{X}(t_1) \), for \( t_1 < t_0 \), then \( A \) must maintain some internal state, i.e. the element is not atomic and must therefore consists of smaller parts. This is illustrated in figure 1.1c. However, even if the physical object \( A \) can be naturally said to consist of some set of objects \( B = \{B_1, B_2, \ldots, B_n\} \) (from some lower level of abstraction, illustrated as green boxes in the figure), the implementer can sometimes still get away with not modeling all of these lower-level objects (the set \( B \)). On the level of abstraction exposed to the user, all aspects that \( B \) causes might not be visible, and the \( A \) can for example be modelled as a finite-state machine, as illustrated in figure 1.1b. The point of doing this is that we hope that it is simpler to implement a finite-state machine than implementing all details of the set \( B \). If the user is only concerned with \( A \) and does not care about the states of the objects in the set \( B \) this is sufficient. But if the user does indeed want to inspect the states of the objects in the set \( B \) – it is not sufficient. How much one can “cheat” therefore depends on what level of abstraction is exposed to the user.

More concretely, and in the context of machine simulation, \( A \) might be the whole instruction execution unit (i.e., \( A \) belongs to a very high level of abstraction). We might not want to inspect the internals of the execution unit, but still care about consequences of the internal aspects, i.e. how \( A \) is implemented in hardware (expressed on some appropriate level of abstraction). Especially, we might want an accurate cycle count after running some set of test programs, for performance measurements. Cycle-accurately modeling the whole instruction execution unit of a real, modern, machine is extremely time consuming. But as in the example with figure 1.1b we might “get away” with only modeling some parts of the internal aspects of the execution unit. If we for example know that (on a very simple processor) the instruction ADD always takes, say, 5 cycles we could simply keep a variable count and add 5 to it each time the execution unit executes the instruction ADD. Similarly for all other instructions, e.g. DIV might always take 15 cycles, and we therefore increase the counter by 15 instead of 5 when a DIV instruction is executed. With this approach we do not get a view of the internals of the execution unit, but we get
1.4 Some requirement’s implementation implications

As one might expect there is significant overlap between how all kinds of (machine) simulators are implemented. There are of course also important special cases, such as if the guest machine shares architecture (ISA) with the host machine, it might be possible to run the guest machine in near-native speed by running the guest programs directly on the host hardware. For example, a x86 guest machine running on a x86 host machine. But this is not the situation considered here – i.e., the guest architecture will not be the same as the host’s. Here, we need techniques such as direct interpretation or dynamic/static (binary) translation. A longer discussion on these techniques is the main focus of chapter 4.

There is also a difference between compiling low-level and high-level languages, and consequently, a difference between simulating low-level and high-level languages. The REPLICA simulator will be given an assembly program to simulate. These differences manifest themselves when expressing a program at different levels of abstractions during compilation (Muchnick 1997), as some information about the program is readily available during some stage, but maybe difficult (or even impossible) to extract during other compilation stages. Furthermore, for this and other reasons, it may be more convenient to express some optimization pass at some specific abstraction layer. Consider the case where a compiler uses multiple internal representations of the program, each at a different level of abstraction. If we are compiling a high-level language, e.g. C, the compilation starts at the highest level of abstraction and the compiler then proceeds by lowering the representation “down” to the next representation format in the abstraction hierarchy, until it reaches the lowest level of abstraction available in the compiler. Between these lowerings, the compiler can do whatever transformation (e.g., some optimization) or analysis that seem appropriate. But, if we (instead) were to compile some low-level language, we start at some program representation lower down in the abstraction hierarchy than in the case where we compile a high-level language. Consequently, we may not have the choice of performing, let us say, some optimization pass at the level that would be most convenient (i.e., some of the higher levels). As some information is lost (some, other, is gained, of course) when lowering from one level to another, if we skip the highest levels (as is the case when

---

8There might, of course, be other reasons why one would do different kinds of approximations. For example, when a machine is only documented in terms of a relatively high-level of abstraction, and no documentation expressed at a lower-level of abstraction is available.
compiling a low-level language) we may not be able to perform all kinds of optimizations available when compiling a high-level language, that starts from the highest level. As a concrete example, consider e.g., the following quote from Kessler [2010]:

Software pipelining has been researched intensively, both as a high-level loop transformation [...] and as low-level optimization late in the code generation process [...]. [...] The latter approaches are bound to fixed instructions and functional units and hence the flexibility of implementing the same abstract operation by a variety of different target machine instructions [...] is lost.

There is also a significant difference between implementing an instruction-accurate simulator and cycle-accurate simulator. Implementing an instruction-accurate simulator has much in common with implementing a programming language, as we in most cases do not care about the hardware at all. We only need to provide the machine architecture’s assembly programming model to the user. Why the programming model looks the way it does is motivated by hardware choices, but we do not need to actually take into consideration how the instruction would be executed on a real machine, as long as we accurately provide the programming model offered by the machine. In chapter 3 we will see all details we need to take into consideration when trying to implement a cycle-accurate (REPLICA) simulator. This has implementation implications as an instruction-accurate implementation has more freedom in how to implement different instructions, whereas a cycle-accurate implementation must execute the instructions in the same way as the guest machine would do. This forced execution pattern might be much more computationally expensive than what can be implemented in an instruction-accurate simulator.

We will see examples (both explicitly and implicitly) of the above observations throughout this thesis.

1.5 Thesis outline

The current chapter has given some (at times vaguely related) background and has introduced the terminology used in the rest of the thesis. However, to not overwhelm the reader, some further terminology will not be introduced until later in the thesis, where appropriate.

To be able to discuss the design and implementation of the simulator, the PRAM model and the REPLICA architecture are defined and discussed in chapter 3. After that, chapter 4 discusses different simulation methods. Simulation of parallel computers and parallelization of these simulation methods are defined in chapter 5. Finally after all these preparation chapters, chapter 6 describes the actual design and implementation of the REPLICA simulator. Lastly, chapter 7 concludes the thesis.

1.6 Summary and conclusion

To sum up and conclude, some appropriate level of abstraction is chosen, and all the below levels of abstraction are declared unimportant for the program’s propose and therefore not modelled. The higher level that is chosen in the abstraction hierarchy, the less time consuming implementation becomes – but the simulation might differ from the real machine in more aspects than a program implemented in terms of a lower level of abstraction. These differences might, however, be of non-importance for the program’s intended use. In the ideal case, one level of abstraction completely hides away all underlying levels. But non-trivial abstractions tend to leak, either by accident, choice (as a design tradeoff) or misuse (by the user of the abstraction). So there might be observable consequences from choosing a higher level, rather than a lower one. Obviously, if the decision where truly orthogonal to all other design decisions we would always chose as an low level as we could think of. But as this cannot be varied independently of e.g. implementation complexity, there are tradeoffs to be made. What ultimately decides what layer is the most appropriate are the (carefully gathered) requirements describing the program’s intended use (and, realistically, time constraints).
Chapter 2

Requirements

This chapter enumerates requirements that describe the REPLICA simulator. Requirements are specified and formatted as follows:

R0 This is an example requirement, to show how requirement specifications are formatted. (Level 1)

Level 1, 2 and 3 are used to specify priority, where 1 denotes highest priority and 3 lowest priority. The terms “should” and “must” are used to specify whether a requirement is optional or not, respectively; in alignment with RFC 2119. All non-optional requirements are (necessarily) level 1, i.e. priority is really only relevant for optional requirements.

2.1 Core requirements and features

The first set of requirements concerns program portability.

R1 The simulator must support both GNU/Linux and Mac OS. (Level 1)

R2 The simulator must be written in C++. (Level 1)

R3 Further, the simulator should, to as large extent as possible and practical, be compatible with the old simulator, e.g. using the same assembly syntax and directives. (Level 1)

The second set of requirements concerns how the user interacts with the program.

R4 A command-line user interface (CLI) for the simulator must be provided. (Level 1)

R5 The simulator should provide some means of debugging programs running on the simulated machine. Such as, displaying register and memory content, single-step execution, breakpoints, and possibly other common debugging features. (Level 2)

The third set of requirements concerns simulator internals and how accurately the REPLICA architecture should be modelled. The terminology used here was defined in section 1.3.2 and partly in the following chapter (the meaning of the terms “step” and “cycle” in the context of REPLICA).

R6 The simulator must be step count accurate. (Level 1)

R7 The simulator should be cycle count accurate. (Level 2)

R8 Step count accurate simulation must be implemented before cycle count accurate simulation (as implied by the priority level), but if cycle count accurate simulation is implemented the simulator must support both modes rather than the latter mode replacing the step count accurate mode. (Level 1)
R9 The simulator should only model internal aspects of REPLICA that are necessary to satisfy requirement \[ R_7 \] (Level 1)

To satisfy requirement \[ R_7 \] the whole network used to communicate between processors and memory modules must be modelled (the network and other components are outlined in the following chapter). This is not needed to satisfy requirement \[ R_6 \]. Implementing the network and all other details needed for step count accuracy requires a non-trivial amount of work, and is therefore marked as optional, as it was not clear in the beginning of the project whether there was going to be enough time to implement all these details.

### 2.2 Extra requirements and features

Some features do not affect the overall architecture of the simulator, in the sense that they can be augmented to an existing simulator without changing the already implemented parts in any non-trivial way. These, so-called, extra features are specified in this section, and because they can, as stated above, be augmented later, all requirements defined in this section are optional.

R10 The simulator should be able to load “architecture configurations” from a configuration file. Such configuration file could include e.g., the number of processors available in the machine and the number of threads per processor. A sound implementation will not depend on that some fixed number of processors (or some other configurable property of the machine) is used in the simulation to work properly, so this does not affect the overall architecture. (Level 2)

R11 The simulator should be able to save and load the memory contents to and from a file through assembly directives. (Level 3)

R12 Aside from the CLI (requirement \[ R_4 \]) there should be a graphical user interface (GUI). (Level 3)

R13 Aside from the CLI (requirement \[ R_4 \]) it should be possible to control the program through only calling it with different flags (such as `sim --cycle-accurate sum.asm`), for e.g. usage in shell scripts. (Level 3)

R14 The simulator should expose an interface to the simulated program allowing pixel-oriented image output. (Level 3)
Chapter 3

The PRAM Model and the REPLICA Architecture

In this chapter the Parallel Random Access Machine (PRAM) model of parallel computation and the REPLICA architecture are introduced. The relation between them is that REPLICA, as previously stated, is an architectural realization of the PRAM model. The reader should, however, when reading this chapter keep in mind that the REPLICA architecture is not a finished product – so the architecture outlined here might differ from the final result of the REPLICA project. Furthermore, the reader should also keep in mind that some references used in this chapter describes REPLICA’s predecessor TOTAL ECLIPSE (or possibly also other, “older”, architectures) rather than the actual REPLICA architecture, which (of course) differs from REPLICA in some (not outlined here) aspects.

The main reference for REPLICA information in this chapter is the unpublished draft Forsell 2011, and the main reference for TOTAL ECLIPSE information is Forsell 2010. Some software conventions, not available anywhere else, are outlined in Åkesson 2012. Almost all content in the REPLICA part of this chapter is based on these articles, i.e. this whole part has the mentioned articles as references. As this is stated here, the mentioned articles are therefore not referred to explicitly in the text.

3.1 The Parallel Random Access Machine (PRAM)

The commonly assumed model of sequential computation is the Random Access Machine (RAM), described in e.g. Keller, Kessler, and Träff 2001 and e.g. used in many introductory books on sequential algorithms, such as Cormen et al. 2009. As the reader might recall, only the most essential parts of real computers are modelled and taken into consideration in the RAM model; that is, e.g., no (as common today) three layers of cache between main memory and the processor, instruction pipeline, instruction-level parallelism or branch penalties.

To extend this from a model of sequential computation to a model of parallel computation one can simply add more processors to the machine. Doing this results in the model called the Parallel Random Access Machine (PRAM). When needed for clarification, the number of processors available in the PRAM under consideration can be explicitly stated, as e.g. “p-processor PRAM”.

---

1 As repeating the same references over and over again would become very verbose, tedious and serve no purpose.
2 Although, one should probably note that the RAM model is not covered in any depth here, because: “Strictly speaking, we should precisely define the instructions of the RAM model and their costs. To do so, however, would be tedious and would yield little insight into algorithm design and analysis.” Still, the important part to note is that it is the RAM model that is used as model of computation.
One important difference from common desktop computers is that in the PRAM model all processors operate synchronously, by sharing a common clock (i.e., all processors execute their instructions in lockstep). This is not true for programming models used on desktop computers, where synchronicity between processors/threads is handled explicitly, by e.g. locks. We will use the terms “time step”, “program step” and “step” synonymously, but avoid the term “cycle” – because it will be used to refer to a different notion in REPLICA. The PRAM processors need not execute the same program. This implies that the PRAM model, in Flynn’s taxonomy, is an instance of a Multiple Instructions Multiple Data (MIMD) machine. Furthermore, as seen directly from the definition above, all processors share a common main memory. Memory writes are directly visible to all processors after a write instruction has been executed (i.e., writes in some step $s$ are visible in the next step $s + 1$). There are no instruction reorderings either. As with synchronicity, this is not true for desktop computers either. So, analogously to the relation between physical sequential computers and the RAM model, the PRAM model is a simplification of the programming models often offered by physical parallel computers.

One problem for the PRAM model, that does not occur in the (sequential) RAM model, is concurrent writes and concurrent reads to one memory location, i.e. more than one processor trying to access the same memory location in some step $s$. We will use “memory access” as a collective term for both reads and writes. There are multiple ways of defining the semantics of concurrent memory accesses, namely:

- **Exclusive read, exclusive write PRAM (EREW PRAM)** At most one processor is allowed to access a memory location $l$ in each step.

- **Concurrent read, exclusive write PRAM (CREW PRAM)** At most one processor is allowed to write to a memory location $l$, but if no processor writes to a memory location $l$ then any number of processors are allowed to read from that memory location.

- **Concurrent read, concurrent write PRAM (CRCW PRAM)** Any number of processors are allowed to either write or read to any memory location in one step. What, then, does it mean for two or more processors to write two or more different values to the same memory location in the same step? There are multiple ways to define the semantics for this as well, to name a few:

  - **Arbitrary CRCW PRAM** No rules are specified, the resulting value for the memory location in the next step is selected in an arbitrary fashion from the set of values written to the memory location in the current step.

  - **Multioperation CRCW PRAM** Some specified arithmetic logical function is performed to “combine” the set of values written to the memory location in the current step, e.g. the maximum or the sum of these values, and the value of the memory location in the next step is the result of this calculation. Multiple functions are available and which is used is selected for each write individually (i.e. there are different write instructions that uses different combine functions). Of course, all processors writing to the same location must use the same combine function.

The naming of the above differs between authors. One should note that regardless of how many processors that access the same memory location (under all the above PRAM models), memory access always takes (only!) one step to complete. This is yet another a property that does not hold for physical parallel machines, as physical memory cannot handle an arbitrary number of request in one cycle.

That “at most one processor is allowed” (in e.g. a EREW PRAM) to do something, or something similar, begs the question: “what happens if more than one processor decides to do that anyway?” This is left to the (software or hardware) implementation to specify. A programmer-friendly implementation might display an error message (in some appropriate way) and a less friendly implementation might exhibit undefined behavior if such an event were to occur.

One question remains to be answered. Consider the following situation: a CRCW 2-processor PRAM is instantiated with each memory location set to 0. Processor 0 will read memory location 0 in the first step, and processor 1 will write 79 to memory location 0 in the first step. What value will processor 0 read? The above discussion
3.2 The REPLICA architecture

does not answer this question in a precise and satisfactory way. (We only stated the guarantee that in the next step memory location 0 will contain 79 – assuming no other processor wrote some other value to this location.)

The answer is as follows. Each step is divided up into three substeps, or phases. Namely, a read substep, a compute substep and, lastly, a write substep. So if a processor reads a memory location in step \( s \) it will see the value from step \( s - 1 \), whether some processor writes to that memory location in step \( s \) does not matter – as this (write) substep would occur after the read substep. One can also use the convention that only one type of memory access at a time is allowed, i.e. there might be multiple readers or multiple writers to one location in one step, but not both (Keller, Kessler, and Traff 2001 uses this convention).

All the above differences between (common) physical parallel computers and the PRAM model raises questions regarding whether it is really possible to efficiently realize the PRAM model in physical hardware – or, equivalently, efficiently “simulate” the PRAM model in hardware. Amongst other concerns, from the above discussion, we see that PRAM offers strict consistency, and as even weaker consistency models such as sequential consistency is, alone, known to restrict common compiler and hardware optimizations (Adve and Gharachorloo 1996; Goetz et al. 2006) these raised concerns might even seem soundly motivated. Nevertheless, there have been attempts to implemented the PRAM model in hardware, e.g. SB-PRAM (Keller, Kessler, and Traff 2001), Explicit Multi-Threading (XMT) (Vishkin 2011) – and, of course, REPLICA, which is outlined in the next section.

For a more thorough treatment of the PRAM model (or “PRAM theory”), relations between the PRAM variants (e.g., the relative strength between variants), complexity measures related to the PRAM model (such as, time, work, etc.), the C-like PRAM language Fork and examples of programs for the PRAM model see e.g. Keller, Kessler, and Traff 2001.

3.2 The REPLICA architecture

REPLICA is, as stated previously, an architectural realization of the PRAM model defined and discussed in the previous section. REPLICA realizes the Arbitrary Multioperation Concurrent Read Concurrent Write (MCRCW) PRAM variant, one of the most powerful PRAM variants. As stated in the previous section, “arbitrary” refers to that if two writes happen in the same step, an arbitrary value of the written values is selected as the result. “Multioperation” refers to the set of instructions that threads can use to cooperate to compute e.g. the sum of a set of values, the minimum of a set of values, and so on (these instructions will be detailed later).

The shared memory inherited from the PRAM model is not implemented directly, but is implemented through Emulated Shared Memory (ESM). Meaning that from the programmer’s perspective the memory looks like one large, shared, memory space, but in reality the memory space is split up into multiple memory modules that communicate with the different processors over a network. REPLICA’s processor, called Multibunched/threaded Architecture with Chaining (MBTAC), is a multi-threaded dual-mode Very Long Instruction Word (VLIW) processor. The default mode, called PRAM mode, is optimized for programs with enough Thread-Level Parallelism (TLP) to hide the latency of the interconnection network used to provide the emulated shared memory. The second mode, called NUMA mode (for, Non-Uniform Memory Access), is an extra mode that can be used to execute program sections with low amounts for TLP where (in NUMA mode) two or more threads are combined into a “NUMA bunch”. The NUMA mode is not discussed in any detail in this thesis, as only PRAM mode will be implemented in the simulator.

A PRAM only have one thread (in the usual sense) per processor, so one does not usually differentiate between these two in the PRAM model. However, REPLICA uses multiple threads per processor. One of the main ideas in REPLICA is that the number of threads per processor and the bandwidth of the network is high enough to hide the network latency, by switching to another thread in the processor when a thread is waiting for a memory request to finish, rather than blocking the execution. This is achieved through the deep instruction pipeline, step caches and scratchpads, which are all outlined in this section.

\(^3\)Which was used as one of the main references for this whole section.
## 3.2.1 A crash-course in REPLICA assembly programming

REPLICA does not necessarily utilize the whole MIMD programming model, rather, each processor always run the same program; which is often called Single Program Multiple Data (SPMD). By introducing control flow that depend on the thread’s id (a unique number assigned to each thread) the SPMD programming model can nevertheless use the whole MIMD model, as illustrated in listing 3.1.

```c
int main() {
    switch (thread_id) {
    case 0:
        program0();
        break;
    case 1:
        program1();
        break;
    // ...
    }
}
```

**Listing 3.1:** MIMD programming model simulation in a SPMD environment.

However, most programs share more code between threads than the program from listing 3.1 so listing 3.2 is more representative of an usual SPMD program.

```c
// Assume that result is a variable
// shared between all threads.
int main() {
    if (thread_id == 0) {
        print("program starting!\n");
    }
    do_important_computations();
    barrier();
    if (thread_id == 0) {
        print("program over, and the result is " + result + "\n!");
    }
}
```

**Listing 3.2:** A program more representative of usual SPMD programs than listing 3.1.

The REPLICA architecture might be described as a parametrized family of architectures, as the architecture is “configurable”. That is, the number of processors, threads per processor, and so on are parameters that can be varied to fit application needs. For example, a REPLICA machine consists of $P$ MBTAC processors, where each processor have $T_p$ threads. (Constituting in total $T = PT_p$ threads for the whole machine.) For each processor there is a dedicated instruction memory and a local data memory module. Each MBTAC processor has $A$ ALUs, which are divided up into pre-memory access and post-memory access ALUs. There are $M$ memory units per processor (together forming a memory module group, i.e. in total $M$ memory module groups), connected to the processors over a high-bandwidth multimesh interconnection network. The case where $M$ is not equal to 1 is out of scope for this thesis.

---

4 Actually, as there is one instruction memory module for each processor, and this restriction does not necessarily hold for the real REPLICA architecture. But this is the approach used in the simulator (and the old simulator as well).

5 Currently, the local data memory is only used in NUMA mode, but this might however change in the future. But as NUMA mode is out of scope of this thesis, the local data memory related details will not be covered here.

6 There are still unresolved questions regarding how this extension will be handled in the architecture, if for example there will be one large memory space or if each memory module group would introduce its own memory space.
3.2 The REPLICA architecture

Because the MBTAC processor is a VLIW processor there are both instructions and subinstructions, where a
instruction consists of one or more subinstruction. In contrast to conventional VLIW processors, MBTAC subin-
structions are allowed to have dependencies between them because MBTAC’s functional units are organized in a
chain-like manner rather than the usual parallel organization. Beyond subinstructions, instructions also consists of
one or more operand. In subinstructions they are referred to as O0, O1, . . . . There are subinstructions for memory
reads and writes, common arithmetical and logical operations, compare operations, jumps and conditional jumps,
and so on. The exact (sub)instruction set for TOTAL ECLIPSE can be found in Forsell [2010].

Each thread has R general purpose registers (e.g., 32), called R0, R1, . . . . There are also temporary result registers,
namely M memory result registers called M0, M1, . . . and A ALU result registers called A0, A1, . . . . The instruction
set is very RISC-like, so all memory accesses are explicit and all non-memory related subinstructions (such as
ALU subinstructions) only operates on registers (such as R0, R7, O0, and so on). The temporary registers content
is only guaranteed to live to the end of the currently executing instruction.

For example, the subinstruction LDn<addr> is used to load the contents of memory location <addr> from
memory module group n. (However, remember that only the case with one memory module group is relevant
for this thesis, so n will always equal 0 here.) For example, LD0 R20 will use the value from register R20 as
address and load the result into the memory result register M0. The REPLICA assembly syntax is fairly simple.
Subinstructions are written in one line with whitespace between them (to separate different subinstructions),
and operands for each subinstruction are separated with a comma (and optional whitespace). The subinstruction
WBn<operand> is (often) used to write the value from some result register (that would otherwise disappear in the
end of the current step) to some general register. For example, WB5 M0 can be used to store the memory read
result into register R5. There is also a store instruction (ST), used to write to memory.

An example assembly program is given in listing 3.3. As seen in the example, assembly directives are used to
allocate space for variables. A variable shared between all threads have a name ending in “_”, e.g. “foo_”. The
memory layout used can be seen in figure 3.1.

```
LBL
  LD0 R1
  OP0 1
  ADD0 R3, O0
  ST0 R3, R1
.data
  .WORD 0x07
```

Listing 3.3: Example REPLICA assembly program.

There are also so-called multioperations and multiprefix operations, used to implement the “multioperation” part
of the REPLICA programming model. As a more in-depth knowledge about the REPLICA hardware is needed
to understand the difference between the different instructions groups from this category, they are addressed in
more detail after the implementation details section that follows shortly.

Now let,

- Ax denote the result of ALU x,
- Mx denote the result of a load from memory module x,
- Ox denote operand x of the executing instruction,
- and Rx denote register x.

Furthermore, let the metavariables Xx and Xy denote any of Ax, Mx, Ox or Rx (from above).

For example, with this notation, memory subinstructions can be presented as in table 3.1.

---

7This might change in the future, to reduce the number of register writebacks.
8To clarify: Xx and Xy are not an actual assembly symbol (i.e., they are never used in actual assembly programs) – theirs only use is for
documentation purposes.
2.2 LLVM

LLVM, Low Level Virtual Machine, is a compiler framework that once started as a research project at the University of Illinois[19]. It has since then grown to become a full fledged compiler framework used in industry applications and now includes front-ends for several languages and back-ends for several architectures[18]. The target version of LLVM in this project is LLVM 3.0.

LLVM front-ends parses and compiles source code into LLVM’s internal representation[18]. LLVM then does target independent optimizations on this internal form[18]. It is then the task of the LLVM back-end to lower this internal representation into machine code[18].

Figure 3.1: The memory layout used in REPLICA, originally figure 5 in Forsell [2011] however, using figure 2.11 from Åkesson [2012] for licensing reasons.
3.2 The REPLICA architecture

### Table 3.1: Simple memory unit instructions.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD&lt;sub&gt;n&lt;/sub&gt; X&lt;sub&gt;x&lt;/sub&gt;</td>
<td>Load word from address X&lt;sub&gt;x&lt;/sub&gt; in memory group n to M&lt;sub&gt;n&lt;/sub&gt;</td>
</tr>
<tr>
<td>ST&lt;sub&gt;n&lt;/sub&gt; X&lt;sub&gt;x&lt;/sub&gt;, X&lt;sub&gt;y&lt;/sub&gt;</td>
<td>Store word X&lt;sub&gt;x&lt;/sub&gt; at address X&lt;sub&gt;y&lt;/sub&gt; in memory group n</td>
</tr>
</tbody>
</table>

To define the set of multiple and multiprefix subinstructions, let z \in \{ ADD, SUB, AND, OR, MAX, MAXU, MIN, MINU \} and see table 3.2. They are paired as follows: 1) M<sub>z</sub>; 2) MP<sub>z</sub>; 3) BM<sub>z</sub> and EM<sub>z</sub>; 4) BMP<sub>z</sub> and EMP<sub>z</sub>; and 5) BMP<sub>z</sub>, SMP<sub>z</sub> and OMP<sub>z</sub>. (Note the double occurrence of BMP<sub>z</sub>.) The differences between these groups will have to wait until after the implementation details discussion.

### Table 3.2: All multioperations and multiprefix operations.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>M&lt;sub&gt;n&lt;/sub&gt; X&lt;sub&gt;x&lt;/sub&gt;, X&lt;sub&gt;y&lt;/sub&gt;</td>
<td>z multiple X&lt;sub&gt;x&lt;/sub&gt; to address X&lt;sub&gt;y&lt;/sub&gt; in memory group n</td>
</tr>
<tr>
<td>MP&lt;sub&gt;n&lt;/sub&gt; X&lt;sub&gt;x&lt;/sub&gt;, X&lt;sub&gt;y&lt;/sub&gt;</td>
<td>Arbitrary multiprefix z X&lt;sub&gt;x&lt;/sub&gt; to address X&lt;sub&gt;y&lt;/sub&gt; in memory group n</td>
</tr>
<tr>
<td>BM&lt;sub&gt;n&lt;/sub&gt; X&lt;sub&gt;x&lt;/sub&gt;, X&lt;sub&gt;y&lt;/sub&gt;</td>
<td>Begin z multiple X&lt;sub&gt;x&lt;/sub&gt; to address X&lt;sub&gt;y&lt;/sub&gt; in memory group n</td>
</tr>
<tr>
<td>EM&lt;sub&gt;n&lt;/sub&gt; X&lt;sub&gt;x&lt;/sub&gt;, X&lt;sub&gt;y&lt;/sub&gt;</td>
<td>End z multiple X&lt;sub&gt;x&lt;/sub&gt; to address X&lt;sub&gt;y&lt;/sub&gt; in memory group n</td>
</tr>
<tr>
<td>BMP&lt;sub&gt;n&lt;/sub&gt; X&lt;sub&gt;x&lt;/sub&gt;, X&lt;sub&gt;y&lt;/sub&gt;</td>
<td>Begin arbitrary multiprefix z X&lt;sub&gt;x&lt;/sub&gt; to address X&lt;sub&gt;y&lt;/sub&gt; in memory group n</td>
</tr>
<tr>
<td>EMP&lt;sub&gt;n&lt;/sub&gt; X&lt;sub&gt;x&lt;/sub&gt;, X&lt;sub&gt;y&lt;/sub&gt;</td>
<td>End arbitrary multiprefix z X&lt;sub&gt;x&lt;/sub&gt; to address X&lt;sub&gt;y&lt;/sub&gt; in memory group n</td>
</tr>
<tr>
<td>OMP&lt;sub&gt;n&lt;/sub&gt; X&lt;sub&gt;x&lt;/sub&gt;, X&lt;sub&gt;y&lt;/sub&gt;</td>
<td>End ordered multiprefix z X&lt;sub&gt;x&lt;/sub&gt; to address X&lt;sub&gt;y&lt;/sub&gt; in memory group n</td>
</tr>
<tr>
<td>SMP&lt;sub&gt;n&lt;/sub&gt; X&lt;sub&gt;x&lt;/sub&gt;, X&lt;sub&gt;y&lt;/sub&gt;</td>
<td>Send multiprefix z X&lt;sub&gt;x&lt;/sub&gt; to address X&lt;sub&gt;y&lt;/sub&gt; in memory group n</td>
</tr>
</tbody>
</table>

3.2.2 Some REPLICA implementation details

Some amount of knowledge about REPLICA’s implementation details is needed to build a simulator (at least a cycle count accurate simulator), and to some extent, also to program it. Therefore, some relevant implementation details are outlined here. An overview of REPLICA’s components are given in figure 3.2. The figure displays many of the components already mentioned, such as processors, memory modules, and the network. As one can see from the figure, there are also switches used to send packets from processors to memory modules (and the other way around as well, for read requests).

**Steps and cycles**

The model of time in the REPLICA architecture directly exposed to the programmer is the same as in the PRAM model, i.e. steps (and memory synchronization conceptually happen between steps). In REPLICA however, there is also the notion of a cycle. A (REPLICA) step consists of more than one (REPLICA) cycle. To clarify, a REPLICA step is the same as a PRAM step, but to efficiently implement the PRAM in actual hardware REPLICA also has the notion of cycles, that divide a step into more than one “phase”. Consider figure 3.3, where the different “phases” are outlined to the right (under “PRAM MODE PIPELINE”). In PRAM mode, one instruction per step is executed, and the instruction’s subinstructions are laid out and executed in the order imposed by the pipeline organization. The pipeline have \( T_p \) stages, i.e. there are as many pipeline stages as there are threads per processor; this is because at each stage in the pipeline (corresponding to a row in the figure), a different thread is located. That is, each row (again, from the perspective of figure 3.3) of the pipeline is used by a different thread. In a cycle, a thread moves from one stage of the pipeline to the next, and a step corresponds to going from the first stage of the pipeline to the last stage of the pipeline.
Figure 3.2: Overview of the TOTAL ECLIPSE architecture (P=processor, M=shared data memory module, L=local data memory module, I=instruction memory module, a=active memory unit, c=step cache, t=scratchpad, and s=switch). Originally figure 3 in Forsell [2010]

Network on chip

One of the biggest problems is hiding memory access latency, and REPLICA addresses this problem in multiple ways. As already seen, figure 3.2 displayed an overview of the REPLICA architecture and its network, but this view is too high-level to implement a cycle count accurate simulator. For cycle count accuracy, all details from figure 3.4 are needed. As seen from the figures, and already mentioned, a network on chip (NOC) is used to communicate between components. There are two (disjoint) networks, one used to send memory access requests (from processors to memory modules), and one used to send read responses back (from memory modules to processors). The two networks are seen in the superswitch in figure 3.4 where one network is directed towards the memory module, and one network is directed away from the memory module. Each of these request and response networks consist of, in turn, one or more (disjoint) subnetworks (submeshes) and which one of these are used to send a packet from e.g. a processor to some memory module is selected in a uniformly random fashion. Actual routing in a submesh is implemented so that a packet is first routed to the correct row, and then to the correct column. The logically shared memory space address is mapped to actual memory modules using a randomly chosen polynomial hashing function. Switch elements in the network must handle collisions, i.e. when more than one of the incoming packets want to go to the same switch element output. There are queues in each switch element for this purpose, and if the relevant queue is full, a busy signal is raised to signal that this has occurred. This is illustrated in the bottom part of figure 3.4. Synchronization waves are used to synchronize the machine’s execution, i.e. to separate between different execution steps, but this mechanism is not outlined in any detail in this thesis.

The previously mentioned step caches are used to reduce the number of packets sent over the network. There is one step cache per processor, as displayed in figure 3.2. Step caches step-wise filter (i.e., filter per step) out all but the first read and first write access from a processor that refer to the same memory location. For writes this is possible because REPLICA realizes the arbitrary flavor of write conflict resolution, and for reads because all reads will get the same result. Consider the situation where five threads, in the same step, from one processor want to write the values 1, 2, 3, 4, 5, respectively, to the memory location 10. Let us furthermore say that the thread that writes the value 1 executes its write instruction first, the processor will take note that some thread has

---

9See e.g., Peh, Kecskler, and Vangal [2009]

10The exact number of subnetworks is decided by another architecture parameter.
3.2 The REPLICA architecture

Figure 3.3: The TOTAL ECLIPSE pipeline, originally figure 4 in Forsell (2010)
Figure 3.4: Overview of a 64-processor TOTAL ECLIPSE network (top), superswitch (middle), and switch element (bottom). Originally figure 11 in Forsell 2010.
written a value to memory location 10 in this step. Now, when all the other threads execute their instructions the processor will see that some other thread already has written the value 1 to this memory location, and simply ignore the write request. So, in total there were five write requests, but because the step cache filtered out all but one, only one message was sent over the network.

Consider now instead the situation where five threads, from the same processor, want to read from the same memory location 40 (again, in the same step, as step caches work step-wisely). When the first read requests is sent, the processor takes note of this (analogously with the write example above). Now, after this first packet was sent away, when the four other threads make their read request to the same location, the processor will see that a read request packet was already sent and make the threads wait for this already sent packet instead of sending another packet. (All packets would return the same value anyway.) So, in total there were five read requests, but because the step cache filtered out all but one, only one message was sent over the network.

Implementing multioperations and multiprefix operations

All REPLICA’s components displayed in figure 3.2 has now been outlined, except the scratchpads and why the memory modules sometimes are called “active” memory units. Scratchpads and active memory units are used to implement multioperations and multiprefix operations. Scratchpads are needed because of the limited associativity in the step caches, and the active memory units are needed for enabling fast computation of the arithmetical/logical operation involved (and to some extent synchronization). Forsell [2006] and Forsell and Roivainen [2011] are important references for the implementation of multioperations and multiprefix operations, and the implementation of ordered multiprefix operations, respectively.

As promised earlier, an outline of the differences between the different kinds of multioperations and multiprefix operations will follow. The result of multiprefix operations are stored in the relevant memory result register, such as M0. The result of multioperations are only stored in shared memory. See table 3.3 for two examples of multioperations.

<table>
<thead>
<tr>
<th>Processor</th>
<th>Thread</th>
<th>T0</th>
<th>T1</th>
<th>T2</th>
<th>T0</th>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ex. 1 values</td>
<td>T0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Ex. 1 results</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>9</td>
</tr>
<tr>
<td>Ex. 2 values</td>
<td>T0</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>Ex. 2 results</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>21</td>
</tr>
</tbody>
</table>

Table 3.3: Two examples where all threads called \( \text{MADD}_0 \ v, \ a \), where \( v \) is each threads respective value from the row called values, and \( a \) is some arbitrary address (but all threads use the same address). The result is only stored in memory.

For an example of a multiprefix operations, see table 3.4. As seen in the example, ordering is lost between processors. Using the notation introduce before, \( \text{BMP}_z \) will calculate the processor-local prefix for all threads that belong to the processor in question. This result is stored in the scratchpad. \( \text{EMP}_z \) will when called then send one packet per processor to the active memory unit (that corresponds to the memory address used) where the total result is calculated and sent back. The result is finally stored in e.g. M0. The instructions from the \( \text{MP}_z \) category does not use the scratchpad and instead sends one packet per thread involved in the operation to the relevant memory unit. There are two variants for multioperations as well. The previous example used \( \text{M}_z \) but \( \text{BM}_z \) and \( \text{EM}_z \) variants are also available. If order is important, there are also ordered multiprefix instructions available (\( \text{BMP}_z \), \( \text{SMP}_z \) and \( \text{OMP}_z \)). These are not outlined in any detail, because they will not be implemented in the simulator. But for a limited number of threads they allow for ordered multiprefix operations.

\[ ^{11}\text{Note that the algorithm presented in this article is incomplete, and additional operations need to be added for a fully working implementation.} \]
linespace

## Table 3.4: One example where all threads called \texttt{BMPADD0} \( v \), \( a \), followed by, \texttt{EMPADD0} \( v \), \( a \) using the same notational convention as Table 3.3, and the values from the result row are stored in each thread's \( M_0 \) register.

<table>
<thead>
<tr>
<th>Processor</th>
<th>( P_0 )</th>
<th>( P_1 )</th>
<th>( P_2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread</td>
<td>( T_0 )</td>
<td>( T_1 )</td>
<td>( T_2 )</td>
</tr>
<tr>
<td>Values</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Scratchpad</td>
<td>3</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>Results</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>

## 3.3 Conclusion

Table 3.5 displays what parts of REPLICA need to be modelled for different levels of accuracy.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Instruction accurate</th>
<th>Cycle count accurate</th>
<th>Cycle accurate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Network</td>
<td>( \times )</td>
<td>( \checkmark )</td>
<td>( \checkmark )</td>
</tr>
<tr>
<td>Separate memory modules</td>
<td>( \times )</td>
<td>( \checkmark )</td>
<td>( \checkmark )</td>
</tr>
<tr>
<td>Step caches</td>
<td>( \times )</td>
<td>( \checkmark )</td>
<td>( \checkmark )</td>
</tr>
<tr>
<td>Scratchpads</td>
<td>( \times )</td>
<td>( \checkmark )</td>
<td>( \checkmark )</td>
</tr>
<tr>
<td>Full model of the pipeline</td>
<td>( \times )</td>
<td>( \times )</td>
<td>( \checkmark )</td>
</tr>
</tbody>
</table>

Table 3.5: Summary of features needed in the simulator for different degrees of accuracy. The symbol \( \times \) means that this feature does not need to be modelled for the associated accuracy level, while \( \checkmark \) means that the feature needs to be modelled.

By “network” is meant essentially the whole network. That is, with packets explicitly modelled, and how they are being sent between switch elements, how they are being buffered in switch element queues and so on. Along with that, a complete implementation of the wave synchronization protocol is needed as well when modelling the network on this level of accuracy. One should also note that, luckily, only cycle accurate simulation needs a full model of the pipeline (i.e., all details from figure 3.3), for the two other levels it is possible to use extremely simplified models of the pipeline (three stages are enough, one for pre-memory, one for memory, and one for post-memory). Furthermore, beyond implementation complexity considerations, there is not sufficient information publicly available to implement a full model of the pipeline (that accurately corresponds to the pipeline used in REPLICA at least).

As can also be seen from the summary table, instruction accurate simulation essentially becomes PRAM simulation but using REPLICA’s instruction set.

Because of these large differences, an implementation that is instruction accurate and one that is cycle accurate do not have much in common. Implementing an instruction accurate simulator has very much in common with implementing a (standard) programming language interpreter, while the implementation of a cycle accurate simulator instead have to focus on how to efficiently simulate the different parts of the computer (that the instruction accurate case can simply ignore). In an instruction accurate simulator, high-level instructions such as multiplications, does not need to be implemented such that they in any way corresponds to how they are executed on a REPLICA machine, as some other (entirely different) execution pattern than that used on REPLICA might be more efficient on the host machine. This allows for more flexibility in the implementation and (hopefully) faster execution speed.

Because of the huge amount of work needed for an cycle (count) accurate simulator, only a instruction accurate simulator will be implemented.
Chapter 4

Survey of Simulation Methods

In this thesis, the situation is as follows. We are given a REPLICA assembly program (that we from now on call guest program) and are to simulate the execution of this program, on usual desktop computers, as if it would run on a (real) REPLICA machine. The survey of simulation methods is not restricted to REPLICA simulation only, but rather concerns machine simulation in general. In the terminology introduced in the previous chapter, the simulator methods considered here regard application virtual machines rather than system virtual machines – as only one program at a time is simulated. In the REPLICA case this distinction is, somewhat, fuzzy as a REPLICA machine does not run a multitasking operating system between the guest program and the hardware.

When it comes to parallel simulation methods, i.e. using more than one thread of execution to speed up the simulation, the discussion will however be REPLICA-specific, as the guest architecture directly affects the simulator architecture and parallelization opportunities. But before parallel simulation methods (which is the main theme of the next chapter, chapter 5), general (sequential) simulation methods are discussed. There are roughly three main classes of general simulation methods:

1. **Interpreters** are discussed in section 4.1 and are the easiest to implement, but result in a much slower simulation execution compared to other methods available.

2. **Static translation** is discussed in section 4.2. The method translates a given guest program to a program for the host architecture (i.e., to a host program) – a method unsuitable for this thesis and therefore not considered in any detail.

3. **Dynamic translation** translates the guest program to a host program “on the fly” in small chunks, and is discussed in section 4.3. This alternative combines interpretation and static translation, thereby avoiding the speed problems inherent in (pure) interpretation, and furthermore many of the problems in (pure) static translation.

Extensions and related methods that do not directly fit into any of these categories are, shortly, surveyed in section 4.4.

As now seen, there are more than one possible architecture to consider when designing a simulator (of any kind) – where possible choices differ in many (important) aspects. Amongst other: How time-consuming it is to implement the architecture? How fast are simulators built according to this architecture? Can all, initially planned, features be implemented when using this architecture, e.g. debugging possibilities? Does the architecture introduce any restrictions on how accurately the simulator can model the real machine?

---

1. This will (hopefully) change in the future.
2. As a side note, we can note that Altman, Kaeli, and Sheffer 2000 used an analogous classification system to describe the three different types of (what they call) “software-based binary translation systems”.

27
There are two main input formats a simulator can receive the guest program in. Either a compiled binary (such as an Executable and Linkable Format (ELF) file) or an assembly source code file. Simulating a program represented by a binary file is more difficult than the case where we have access to an assembly source code file, especially for static translation methods (discussed in further depth in section 4.2). This thesis mainly focuses on the case where we are given an assembly source code file, but considers where needed the binary file case as well as the simulator might, in future work, be extended to handle this type of input.

As a concluding remark for this introduction section, retargetable simulation (Brandner, Horspool, and Krall 2010) using e.g. Processor Description Languages (PDL), also called Architecture Description Languages (ADL), is not considered in this thesis because its irrelevance to the goal of the thesis (beyond requirement R10). PDLs are “just” another layer of indirection, even if the description language and accompanying tools are capable of (semi-)automatically generate simulators, it is just the same set of simulation methods that are available to hand-written simulators that are available to generated simulators.

4.1 Direct interpretation

Direct interpretation is partly of interest in itself because it is significantly simpler (Farfeleder, Krall, and Horspool 2007; Shan et al. 2011; Ertl and Gregg 2003) to implement than the two other methods, and partly because it is needed as a “support mechanism” for other simulation methods (Ertl and Gregg 2003) (discussed in further detail in relevant sections later). But is, as we will see, significantly slower than other methods (Farfeleder, Krall, and Horspool 2007; Shan et al. 2011; Jones 2010; Smith and Nair 2005).

There are many ways of dividing up interpreters into different categories. One important categorization is what internal representation (IR) is used when interpreting. In high-level language interpretation we are given the actual high-level language to start with, rather than an assembly program as in our case. We might then directly interpret the program, i.e. walking the abstract syntax tree (AST) – as done by e.g. (at least older versions) of the main Ruby implementation. An alternative solution is to first (statically) compile the program to some low-level representation, an assembly-like language often called bytecode, and then execute the program in its new representation. That is, the interpreter interprets the bytecode rather than the high-level language directly (which is often more efficient (Ertl and Gregg 2003)), as done in e.g. CPython, Lua, and the OpenJDK JVM (the footnotes gives details about each implementation’s bytecode interpreter). Only the later kind is relevant for this thesis, and the first kind is therefore not discussed further. Moreover, in our case we get the assembly code directly, and in this sense we get the first step (i.e., translating from an AST to bytecode) for free. As an example of how bytecode might look like, in this case for CPython, see listings 4.1 and 4.2.

```python
from dis import dis
def foo():
    a = 4*b + c/d
    print a
dis(foo)
```

Listing 4.1: The small example Python program bytecodeprint.py, to be translated into bytecode.

---

3 Older versions of the Ruby reference implementation worked by traversing the AST directly (Sasada 2005), but as of Ruby 1.9, Yet Another Ruby VM (YARV) is used which first compiles the Ruby code to bytecode and then interprets it, like the implementation examples that follows.

4 http://hg.python.org/cpython/file/58b1d93c933b/Python/ceval.c#l824
   http://www.lua.org/source/5.2/lvm.c.html using a very simple interpreter because of portability concerns (Ierusalimschy, Figueiredo, and Celci 2005). The other projects listed here solved this portability problem through extensive macro usage.

6 http://openjdk.java.net/groups/hotspot/docs/RuntimeOverview.html#Interpreter|outline and
   http://hg.openjdk.java.net/jdk8/jdk8/hotspot/file/905945c5913e/src/share/vm/interpreter/bytecodeInterpreter.cpp

---
4.1 Direct interpretation

> python --version
Python 2.7.2+
> python bytecodeprint.py

<table>
<thead>
<tr>
<th>n</th>
<th>Operation</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>LOAD_CONST</td>
<td>1 (4)</td>
</tr>
<tr>
<td>5</td>
<td>PRINT_ITEM</td>
<td>0 (a)</td>
</tr>
<tr>
<td>6</td>
<td>LOAD_GLOBAL</td>
<td>0 (b)</td>
</tr>
<tr>
<td>7</td>
<td>BINARY_MULTIPLY</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>LOAD_GLOBAL</td>
<td>1 (c)</td>
</tr>
<tr>
<td>9</td>
<td>BINARY_DIVIDE</td>
<td>2 (d)</td>
</tr>
<tr>
<td>10</td>
<td>LOAD_GLOBAL</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>BINARY_ADD</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>PRINT_NEWLINE</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>LOAD_CONST</td>
<td>0 (None)</td>
</tr>
<tr>
<td>14</td>
<td>RETURN_VALUE</td>
<td></td>
</tr>
</tbody>
</table>

Listing 4.2: Bytecode for program in listing 4.1.

The OpenJDK JVM also dynamically compile the bytecode to native code, which is a simulation method discussed later in this thesis (section 4.3). Other optimizations that can be found in the mentioned interpreters are discussed in section 4.1.2.

4.1.1 A basic interpreter for a simple machine

As stated earlier, direct interpretation is simple to implement – as will be demonstrated here by providing a fully functional implementation of a small and simple machine model. The machine has three registers (r1, r2, r3) and a small set of instructions (ADD, SET,...). Further details are given in the source code provided. The overall program structure is described by listing 4.3, output from the example programs embedded in the source code are displayed in listing 4.4, unimportant implementation details are found in listing A.1 on page 71.
Chapter 4. Survey of Simulation Methods

const instruction_t program1[] = {
  /* 0: set r0, 0 */
  { SET_OP, &regs.r0, 0, 0, 0, 0},
  /* 1: set r1, 1 */
  { SET_OP, &regs.r1, 0, 0, 0, 1},
  /* 2: set r2, 10 */
  { SET_OP, &regs.r2, 0, 0, 0, 10},
  /* 3: add r0, r1, r0 */
  { ADD_OP, &regs.r0, &regs.r1, &regs.r0, 0, 0},
  /* 4: display r0 */
  { DISPLAY_OP, &regs.r0, 0, 0, 0, 0},
  /* 5: cmp_eq r0, r2 */
  { CMP_EQ_OP, &regs.r0, &regs.r2, 0, 0, 0},
  /* 6: jump_if_true 8 */
  { JUMP_IF_TRUE_OP, 0, 0, 0, 8, 0},
  /* 7: jump 3 */
  { JUMP_OP, 0, 0, 0, 3, 0},
  /* 8: halt */
  { HALT_OP, 0, 0, 0, 0}
};

int main() {
  // Initialize some of the registers, e.g. pc = 0.
  boot_machine(regs);

  // The main dispatch loop,
  // this is where we fetch, decode and execute instructions.
  while (get_next_instruction(program0, regs, inst)) {
    // Proceed to the next instruction,
    // may be overwritten by jumps later on in this cycle.
    ++(regs.pc);

    switch (inst.op) {
    // add <r0>, <r1>, <r2> =>
    // <r2> = <r0> + <r1>
    case ADD_OP:
      *(inst.r2) = *(inst.r0) + *(inst.r1);
      break;
    // set <r0>, <const> =>
    // <r0> = <const>
    case SET_OP:
      *(inst.r0) = inst.val;
      break;
    // cmp_eq <r0>, <r1> =>
    // cond true iff r0 equal to r1
    case CMP_EQ_OP:
      regs.cond = (*(inst.r0) == *(inst.r1));
      break;
    }
  }
}
// display <r0> =>
// prints <r0> to stdout
    case DISPLAY_OP:
        std::cout << *(inst.r0) << std::endl;
        break;

// jump <addr> =>
// <pc> = <addr>
    case JUMP_OP:
        regs.pc = inst.addr;
        break;

// jump_if_true <addr> =>
// if <cond> = true, then <pc> = <addr>
    case JUMP_IF_TRUE_OP:
        if (regs.cond)
            regs.pc = inst.addr;
        break;

// halt =>
// halts the machine
    case HALT_OP:
        regs.halted = true;
        break;
    }
    }
    print_halt_message(regs);

Listing 4.3: Overall structure of a simple direct interpreter.

Output from program0:
2
3
The machine has halted, its final state being:
r0 = 1, r1 = 2, r2 = 3
cond = 0, pc = 7

Output from program1:
1
2
3
4
5
6
7
8
9
10
The machine has halted, its final state being:
r0 = 10, r1 = 1, r2 = 10
cond = 1, pc = 9

Listing 4.4: Output from program0 and program1, respectively, from 4.3

Adding more instructions to the interpreter is simply a matter of adding yet another case in the switch statement.

One difference between the example program in listing 4.3 and the REPLICA simulator is that in the above example the guest program is encoded directly in the host program rather than being read from a file. Transforming the guest program represented by the assembly file to a similar IR as used here could be added as an extra preprocessing step. When using an assembly input representation, this preprocessing step is fairly straightforward.
But for the binary case, this is not a trivial as it might sound. This is not a problem unique to interpretation, but rather, is applies to all three simulation methods. Because static translation have most problems with this it is discussed in section 4.2.

### 4.1.2 Problems with direct interpretation

The main problem with direct interpretation is that it is slow, as stated above. At least for simulators with a low-level instruction format the instruction dispatch introduces significant overhead; the problem is less significant for simulators with a more high-level instruction format (Brunthaler 2009). High-level instruction formats might, for example, involve polymorphism and operations that cannot be directly implemented in hardware (such as arbitrary-precision arithmetic). A simulator’s instruction format can, of course, contain both high-level and low-level instructions. Brunthaler’s PhD thesis (Brunthaler 2011) expands on this topic, but is (it seems), mostly directed towards optimizations for so-called high abstraction-level virtual machines (e.g., an inline caching technique based on quickening, partial stack-frame caching, eliminating redundant reference count operations in immediate reference counting, ...). From the author’s limited experience, this seems to hold in general as well. That is, many (or, even, most) optimization techniques seems to rely on high-level notions being available (see section 4.3), and does not apply to the situation where only an assembly representation of the program is available. Furthermore, as Brunthaler notes in the introduction of his thesis, “Improving the performance of virtual machines has been a topic of considerable interest during the past 25 years, and continues to be an active research area until today. [...] However, most of the research has focused almost exclusively on improving the performance of dynamic compilation sub-systems.” Duesterwald (2007) in the context of dynamic compilation, notes a related problem. She claims that one important difference between just-in-time and binary dynamic compilation, where we are only given a compiled binary, is the different levels of abstractions in their input they receive. The important conclusion for this thesis is that “the availability of semantic information in a JIT compiler allows for a larger optimization repertoire than binary dynamic compilation.”

For the REPLICA simulator, we are, indeed, given a low-level representation of the program, and consequently there are not many optimizations (to the author’s limited knowledge) available. One important exception is the multioperations and multiprefix operations, which can be implemented in a way that does not correspond to how they are implemented in the actual REPLICA machine. This is because prefix sums and so on are fairly high-level concepts, so there is still some choices left to be made about how they are implemented (i.e., the compiler that generated the code have not done the choice for us already). But this is not limited to interpretation (and is therefore discussed in the chapter on parallel simulation, chapter 5).

Regardless of all of the above, it still holds that for each guest instruction, more than one host instruction is executed. If we would have e.g. 2 host instructions per guest instruction, we could end up with a slowdown of (at least) 2, with 10 host instructions per guest instruction we will get a slowdown of (at least) 10, and so on. This is because we need to simulate, in software, much of what the guest machine could do in hardware, e.g., amongst other, updating the condition register (if such a thing exists on the guest machine). How much overhead this introduces depends on how much the guest and host machine differ in architecture. For example, one could expect that simulating a 32-bit machine on a 8-bit machine involves some overhead as one has to implement 32-bit arithmetic in 8-bit arithmetic (shifting bits back and forth, handling overflows correctly and so on). Another example being that the floating-point format might differ – software implementations of floating-point arithmetic are not exactly known for theirs lightning-fast execution speed. But on other hand, there isn’t much the other simulation methods can do about this problem either.

With the above discussion in mind, as the interpreter largely consists of one giant switch statement we will first consider what switch statements are compiled down to, and after that do a (very) short survey of the (limited) set of optimizations available to interpreters for low-level program representations.

---

7That is used to mean runtime compilation of intermediate virtual machine code, such for e.g. Smalltalk-80 and Java.
4.1 Direct interpretation

Code generation for switch statements

There are multiple ways of generating code for switch statements (Aho et al. 2007; Korobeynikov 2007).
Consider listing 4.5. Let $C$ denote the set \{C_1, C_2, \ldots, C_n\} and let $n = |C|$, the cardinality of $C$. For presentation simplicity, $C$ will always only consist of integers.

```c
switch (E) {
    case C_1: S_1; break;
    case C_2: S_2; break;
    /* ... */
    case C_n: S_n; break;
    default: default_statements;
}
```

Listing 4.5: The switch statement syntax.

For small $n$, the switch statement can be implemented as a sequence of conditional jumps. If the elements of $C$ are distributed in such a way that they cover a large enough amount of some interval $[i_{\text{min}}, i_{\text{max}}]$ the switch statement can be efficiently implemented as a jump table. A switch implemented as a jump table has time complexity of $O(1)$ and an implementation that compares one integer at a time is $O(n)$ \(^4\) (This overhead can be reduce somewhat, to $O(\log n)$, by using a binary search strategy.) As for space complexity however, the jump table have $O(i_{\text{max}} - i_{\text{min}})$ while the conditional jumps solution have $O(n)$. If $n$ do not satisfy $n \approx i_{\text{max}} - i_{\text{min}}$ the jump table will waste space. If this condition is not satisfied, and space complexity is important, the switch statement can be implemented as a hash table (with, as we know, an average time complexity of $O(1)$ and a worst-case complexity of $O(n)$). As we know $n$ statically, the space complexity becomes $O(n)$. (Actually, as we know the content of $C$ statically as well, this information could maybe be used to somehow select an appropriate hash function.) More alternatives are outlined in Korobeynikov 2007.

For simulators we actually get the best possible case here, at least if the opcode numbering is only used internally – so we do not have to use some carefully chosen future-safe opcode numbering to guaranty ABI compatibility between versions. That is, the opcodes can always be in the interval $[0, n - 1]$ (without any “holes”).

For example, the switch statement in the example interpreter (listing 4.3) is, on a x86_64 machine with a recent version of GCC, implemented as a jump table, as shown in the assembly fragment in listing 4.6.

```assembly
.L31:
    mov inst(%rip), %eax
    jmp *+.L20(,%rax,8)
.L20:
    .quad .L8
    .quad .L10
    .quad .L11
    .quad .L12
    .quad .L30
```

Listing 4.6: A jump table implemented in x86_64 AT&T-style assembly.

(Micro-)optimizations to reduce the interpretation overhead

Ertl and Gregg 2003 claims that interpreters “perform an exceptionally high number of indirect branches”, and furthermore claims that this is not necessarily obvious for everyone, and that for example “even the most promi-

---

\(^4\)If we, for some reason, know (or think we know) the probability distribution of the elements of $C$, a well-known micro-optimization is then to put the case with the highest probability first in the switch, the case with the second highest probability in second place, and so on – this might reduce the actual running time (depending on how uneven the probability distribution is, and assuming the switch is implemented as a sequence of conditional jumps).
The paper on interpreter performance characteristics (Romer et al. 1996) misses this point. Consequently, interpreters are highly dependent on the indirect branch predictor used, with speedup factors up to 4.77 between no predictor and a good predictor. They also do various measurements using existing interpreters with different indirect branch predictor schemes and mispredict penalties. They conclude that "the most useful hardware support feature for efficient interpreters is good indirect branch prediction" and give some advice on how to write efficient interpreters. See figure 4.1 for an illustration of this problem.

```c
while (!halted) {
    switch (opcode) {
    case ADD:
        /* ... */
        break;
    case SUB:
        /* ... */
        break;
    /* ... */
    case MULT:
        /* ... */
        break;
    }
}
```

**Figure 4.1:** Control-flow for a `switch` statement, an indirect branch predictor will have problems to predict the branch target for the green arrow (the branch marked with a question mark).

Casey, Ertl, and Gregg (2007) suggest two optimizations (replicating instructions and combining sequences of instructions into so-called superinstructions) to reduce the indirect branch misprediction overhead, at least in the context of branch target buffers (BTBs) indirect branch prediction (the most common form of prediction when the paper was published). They achieve speedups up to 4.55 over common (threaded) interpreters. Davis and Waldron (2003) provides a (very) short survey of optimizations available for interpreters.

Ertl and Gregg (2003) claims direct threaded code (Bell 1973) to be the most efficient dispatch method. One problem however is that this method cannot be implemented in (standard) C. But compiler-specific extensions, available e.g. in GCC and Clang make it possible to implement it in (this augmented version of) C. Listing 4.7 shows an example of how the resulting interpreter can look (compare this to listing 4.3 using the switch-based approach). As mentioned in the introduction of this section, with the help of macros and conditional compilation the interpreter can be written in such a way that it utilizes direct threaded code when the needed compiler extensions are available, and falls back to the traditional approach when they are not. With direct threaded code, the program (i.e., the internal representation of the program) becomes a sequence of instruction locations.

---

9 Using superinstructions in a simulator for REPLICA is not (at least without any extra work involved/directly) possible because one cannot move subinstructions over instruction boundaries without risking breaking the memory model that the programmer expects, and consequently, one cannot combine two or more instructions into superinstructions.


in listing 4.7), whereas in the traditional case the program consists of a sequence of opcodes (e.g., `program0` in listing 4.3). As the “direct” in “direct threaded code” suggest, there are multiple kinds of threaded code. For example, indirect threaded code (Dewar 1975), which introduces an additional layer of indirection. Token-switch threaded code is another common variant. See e.g. Klint (1981); Smith and Nair (2005) for an comparison between the (seemingly) most common variants (which are, if fact, those mentioned here).

```c
void *program[] = { &&add, /* ... */ };

// As this is a stack machine,
// instruction operands are implicit.
int main() {
    void **ip = program;
    int *sp;
    goto *ip++;
    add:
    sp[1] = sp[0] + sp[1];
    sp++;
    goto *ip++;
    /* Rest of the instructions .. */
}
```

Listing 4.7: Program based on figure 1 in Ertl and Gregg (2003) “Direct threaded code using GNU C’s labels as values”.

Additional optimization approaches have been tried, such as software pipelining-like techniques (Smith and Nair 2005). As a last remark, in all examples this far, we where given the program in some convenient format that did not need and extra decoding to access it. Assume that this is not the case. Decoding overhead can be reduced by caching the decoded result in some form that is computationally easier for interpreter to use. When interpreting assembly programs the decoding overhead is in the form of parsing, and when interpreting binary programs (i.e., programs compiled to executable files) the overhead is in the form of shifting and masking. This cache, or simply IR, can be built before program execution, at program start up, or during program execution.

### 4.1.3 Debugging

Implementing debugging facilities such as single stepping, and displaying and modifying the state of the machine is trivial with this simulation technique (Farfeleder, Krall, and Horspool 2007). For single stepping, simply give back the control to the user after executing the `switch` statement (or equivalent implementation). To display machine state, as e.g. registers, simply print the contents of the relevant `struct`.

### 4.1.4 Summary and conclusion

A direct interpreter is simple and easy to implement, but slow. It seems difficult to significantly improving the execution speed, but at least some standard (micro-)optimizations are available.

### 4.2 Static translation

As now seen, direct interpretation is expensive timewise, as much extra work is done during runtime. The obvious question then being if this somehow can be done statically instead. That is, statically translating (compiling) the
guest program from a REPLICA program to a program native for the host architecture, so the program can execute as any other host program would do – without any extra jump overhead between all guest instruction.

A simulator using static translation is in most cases faster than direct interpretation, but as we shall see this method has its own disadvantages. Although faster than interpretation, static translation from a compiled binary or assembly source code does in general not achieve as good code output as if the program were compiled from high-level language, as the compiler receiving the high-level input can “view the target program in its entirety and at a higher level of abstraction”, which enables the program to better optimize the program (Jones 2010).

Not relevant for our situation, but may be in future extensions of the REPLICA simulator, is that problems arise when a binary program is given as simulator input (rather than an assembly file). Even locating the code in the binary is a non-trivial task – a seemingly necessary task if the simulator should have a chance of translating it. For a large class of machines, the problem of locating code is even undecidable. This holds true for amongst others the x86 architecture (Wartell et al. 2011). Consequently, in the related area of disassembling binary files, well known programs such as GNU objdump uses a very simple heuristic (Eagle 2011), and (of course) more advanced programs such as IDA Pro only disassemble heuristically as well (Eagle 2011; Wartell et al. 2011).

One problem is that code and data may be mixed, e.g. because that the compiler that produced the binary might have padded the instruction stream to align jump targets (Smith and Nair 2005), or the binary might stem from a handwritten assembly program (and who knows what an assembly programmer might come up with).

However, when given an assembly source code file representation of the program as input (as is the case for the REPLICA simulator), the problem of locating code simply goes away, as we can (simply) look, without considering context, on a source code line to see whether it is code or not. There are of course special cases in static binary translation (i.e., when we are given a binary as input) that we could handle. If we for example, for some reason, know that data and code will not be mixed and therefore a simple linear instruction-by-instruction decoding will yield valid instructions.

As stated, problems in static binary translation do not affect us currently as we are given an assembly source file. But relying on this would restrict further extensions to the simulator, i.e. support for program binaries as input.

Simulating self-referential code poses problems, and self-modifying code is not possible to handle with pure static translation (Altman, Kaeli, and Sheffer 2000; Jones 2010). Using a “fallback” interpreter is one possible way around this problem (Cifuentes and Malhotra 1996; Farfeleder, Krall, and Horspool 2007; Jones 2010). Self-modifying code occurs in e.g. operating systems loading executables and programs utilizing dynamic compilation, because in both cases guest-native code is generated that cannot, in general, be inferred during static translation. Requiring two types of simulation methods is of course unwanted implementation overhead.

Compared to direct interpretation, it is also less clear how single stepping, breakpoints and other debugging techniques would be (efficiently) implemented. Farfeleder, Krall, and Horspool 2007 solved this by relying on direct interpretation as a backup.

It is not needed to implement a whole compiler back-end to implement static translation, as e.g. C code can be generated, given a guest assembly file as input, and then compiled with an existing compiler for the host system, such as GCC. This method is often referred to as compiled simulation (Farfeleder, Krall, and Horspool 2007; Brandner, Horspool, and Krall 2010).

### 4.2.1 Summary and conclusion

Static translation is in most cases faster than direct interpretation. But because of its problems with binary input files and difficulties with implementing debugging techniques for this simulation method, static translation will not be used to implement the REPLICA simulator.
4.3 Dynamic translation

With direct interpretation one instruction at a time is (in some sense at least) translated, and in static translation the whole program is translated in one go. These being two extremes, one might wonder if there is some simulation method that translates larger than one instruction-sized but smaller than whole-program-sized units of the program at a time. It, of course, does exist and is called dynamic translation.

Dynamic translation is not as straightforward as direct interpretation, but offers significant speedup in comparison to purely interpretive simulation methods (Jones 2010). Dynamic compilation refers to runtime generation of executable code (Duesterwald 2007). But we will not differentiate between just-in-time (JIT) compilation and dynamic translation, as e.g. Duesterwald (2007) does. Often, instead of saying “JIT compiling” we say “jitting” (i.e., without any special capitalization), and sometimes even just “jit”. Jitting will be the preferred term as it is shorter than “dynamic translation”.

Dynamic compilation (in the general sense) has applications in a much wider scope than considered here, and can with the help of runtime information solve problems that static compilation can only solve partly, or not at all. For example, reducing the overhead introduced by extensive use of polymorphism as encouraged in object-oriented programming, or optimizing across dynamic bindings, introduced by e.g. dynamically linked libraries. How a JVM implantation can reduce the cost of polymorphism, by techniques not available in static compilation, is provided as an illustrative example of runtime information utilization later in this chapter – but before that, a discussion on the basics of jitting is needed.

There are essentially two degrees of freedom when jitting: (i) What to jit, here referred to as translation unit (TU), i.e. how large program fragments at a time are compiled, and (ii) when to jit (i.e., when to translate the TUs). In short, instead of directly interpreting the original (guest) program as we did in section 4.1, when jitting we instead generate (host) native code, followed by later executing it.

4.3.1 What to translate

When jitting high-level languages such as Java where the notion of classes and methods are available, boundaries induced by these may be used to define TUs. For more low-level languages, as is more relevant for this thesis, such as some assembly dialect, or even compiled binaries such as ELF files, dynamic basic blocks may be used instead. Because of, amongst other, (register) indirect branches, finding static basic block boundaries becomes undecidable. Here, “static basic blocks” refers to basic blocks in the usual sense. On other hand, a dynamic basic block depends on actual control flow, and begins at an instruction immediately following a jump and ends with the next jump (Smith and Nair 2005, p. 57). So, in comparison to (static) basic blocks, dynamic basic blocks might contain instructions that are jump targets and might overlap with other dynamic basic blocks. In this thesis, in the context of jitting, when we say “basic block” we mean “dynamic basic block”.

Despite that dynamic basic blocks tend do be longer than static basic blocks (Smith and Nair 2005, p. 57) they are too short to use as TUs for efficient simulation. Sequences of basic blocks, so called traces, is one possible TU that can be used to go beyond basic block boundaries.

Because one can translate the program bit by bit, dynamic translation does not suffer from the same code discovery problems that static translation does.

4.3.2 When to translate

As regards to when jitting might occur there are multiple answers as well. Jitting is not in any way forced to occur the first time a TU is to be executed. The jit system might for example choose to interpret the TU the first, say, 63 times it is executed, but when the 64:th invocation occurs, jit the TU and in the 64:th and later invocation execute the (host native) code generated in the jitting process. Not jitting on the first invocation has multiple advantages.
First of all, compilation is a time consuming process and if the TU is not executed often enough after the jitting has occurred, it might, in total, be slower to jit a TU than to only directly interpret it. Waiting some number of times before jitting the TU is meant to reduce the chance of this happening, as if we have seen the TU being executed many times before, we expect it to be executed many times in the future as well. This is an example of (simple) profile-based jitting, i.e. utilizing runtime information to guide translation. There are of course more advanced profile measurements than just counting the number of times the TU has been invoked.

Alternatively, a jit system might jit without any, or very little, optimization on the first TU invocation, and when the TU has been invoked some number of times, recompile it, but this time with more aggressive optimization methods enabled, and if the TU is very heavily used recompile it again with even more aggressive optimization. That is, a jit system can be built to adapt to the runtime information available. This is called adaptive optimization, or adaptive jitting. However, one should remember than if the jit system only has access to a low-level representation of the program, the jit compiler cannot perform many of the usual optimizations used when compiling programs represented in high-level languages (Altman, Kaeli, and Sheffer 2000; Duesterwald 2007), as already noted previously.

### 4.3.3 Simple just-in-time compilation

Many jit systems share the architecture outlined in figure 4.2 (Shan et al. 2011; Böhm, Franke, and Topham 2010; Jones 2010; Duesterwald 2007; Smith and Nair 2005). Of course, some deviate more than others, e.g. Böhm, Franke, and Topham 2010 as no epochs are drawn in figure 4.2.

![Control flow for a jit system, when using an interpreter.](image)

**Figure 4.2:** Control flow for a jit system, when using an interpreter.

We will assume that the jit system uses direct interpretation and only compiles “hot spots”, i.e. TUs that the system has predicted will be executed frequently in the future (by some implementation-defined heuristic). The situation where no interpreter is available is similar.
After a TU has been translated it is stored in a code cache to reduce the overhead of translating the same code multiple times. Before an instruction is executed, thejit system checks if the associated TU is available in the code cache, otherwise it has to decide whether to directly interpret the instruction, or to compile the TU starting at the current instruction (as stated before, runtime information can be used to guide the decision).

We want as long TUs as possible to reduce the context-switch overhead when jumping between the dispatch loop and compiled TUs. The guest registers can be saved in a record as in direct interpretation, i.e. listing 4.3’s registers_t. Each TU needs an entry stub to possibly load guest register content (from memory cache, hopefully, or in worst case main memory) into host registers. Each TU also needs an exit stub that writes back the guest register content to the record, to free host registers for other uses (as they might be needed in the dispatch loop) and make the guest registers content available to subsequent TUs and the interpreter.

4.3.4 Advanced just-in-time compilation

More advanced jit techniques are beyond the scope of this thesis, and many rely on the presence of notions that are not easily determined from assembly code alone (Duesterwald 2007), such as method boundaries or virtual method calls, as (again) already discussed. Instead of doing an in-depth discussion of how different jit techniques may work, we present an example of how a JVM implementation utilizes runtime information when doing the jit compilation.

Goetz 2004 discusses some of the optimization techniques used in the HotSpot JVM that have no direct counterpart in static compilation. One of them is dynamic deoptimization. Method inlining is not entirely straightforward in object-oriented languages as a method m call on a object of type A might, in fact, call some derived class B’s method m. Statically identifying the call target is undecidable, and the compiler has to do with conservative points-to analysis. Dynamically, however, more information might be available and allows for further optimizations. Among others, which classes are loaded by the system. If none of the loaded classes inherits from A we know that the method call A.m will in fact call A.m (and not some other method B.m), and the compiler can therefore safely inline the method, or perform a monomorphic call transformation that translates the virtual call to a direct call.

The problem is that this might not hold in the future. After all, new classes can still be loaded after the TU compilation. If in fact a class B that inherits from A is loaded the assumption used when compiling no longer holds, and the TU has to be deoptimized, either by recompilation or by removing the TU from the code cache (and instead waiting for the TU to become hot enough before compiling it again).

4.3.5 Practical aspects: code generation and jit frameworks

Building a compiler back-end that is both fast and generates decent code is, to say the least, a non-trivial task. A, what is here called, “jit framework” could be used to ease some of this non-triviality. There are a few projects that can be used as a jit framework, amongst others libjit Nanojit GNU Lightning and LLVM. Both libjit and GNU Lightning have reasonable documentation coverage but the latest stable release of libjit is from 2008 and the latest stable release of GNU Lightning is from 2004. Nanojit is used in the SpiderMonkey jit (which is used in the browser Firefox) but has fairly limited documentation. For example, how to compile it is “left as an exercise for the reader”. Which is sad, as it shows some promise Nanojit, libjit and GNU Lightning could all be considered smaller projects, and in comparison to larger projects, such as LLVM, generally generate code
faster—this is partly a consequence of spending less time on code optimization (but thereby also generate code of lower quality). For example, Meurer [2010] found that LLVM have “significant compilation overhead”, and notes that other projects have experienced the same thing. Finding comparisons between the projects turned out to be difficult, and those that were found were possibly outdated as they were written some years ago, or highly biased.

All four mentioned JIT frameworks provide a similar programming interface, outlined in listing 4.8.

```c
// Excerpt from tutorial/t1.c from libjit-0.1.2.tar.gz.
params[0] = jit_type_int;
params[1] = jit_type_int;
params[2] = jit_type_int;
signature = jit_type_create_signature(
    jit_abi_cdecl, jit_type_int, params, 3, 1);
function = jit_function_create(context, signature);

x = jit_value_get_param(function, 0);
y = jit_value_get_param(function, 1);
z = jit_value_get_param(function, 2);
temp1 = jit_insn_mul(function, x, y);
temp2 = jit_insn_add(function, temp1, z);
jit_insn_return(function, temp2);
jit_function_compile(function);

// The function can now be called with jit_function_apply.
```

Listing 4.8: Example of how the above mentioned JIT frameworks are used. For this specific example libjit was used, but the other can be used in a similar way.

### 4.3.6 Related work

For a brief overview of JIT research and usage between 1960 to 2000 see Aycock [2003] Arnold et al. [2005] surveys more recent work.

Wennborg [2010] discusses code-generation improvements for an existing JIT-compiling simulator using LLVM. Joloboff et al. [2011] discusses JIT-compilation using LLVM for a simulator as well. Both translate a basic block at a time, represented as a LLVM function. Joloboff et al. [2011] uses a slightly more advanced approach, as the simulator first simulates by direct interpretation, and basic blocks are JIT-compiled only after being executed some specified number of times (each basic block is associated with a counter to keep track of this number).

Böhm, Franke, and Topham [2010] presents yet another JIT-compiling simulator, but in contrast to other fast simulators their simulator also models microarchitectural aspects such as the instruction pipeline and memory hierarchies, resulting in almost cycle count accuracy. In one benchmark the average cycle count prediction deviation was 1.4% compared against using “calibrated interpretive cycle simulation mode”. How this is achieved is briefly outlined.

Arvedahl [2002] discusses, mainly, code generation for the Intel Itanium architecture for Kaffe, a relatively simple Java Virtual Machine (JVM) implementation. The JIT architecture, which is not treated in any depth in the thesis, works in such a way that one Java method at a time is compiled. Compilation occurs when the method is called for the first time. Arvedahl concludes that an optimizing JIT compiler for the Itanium architecture should detect portions of the code that are frequently executed and spend more time optimizing these portions, and use a code generator that generates less optimized code but is faster for the rest of the code (Arvedahl [2002] p. 50).

---

4.3.7 Debugging

For jit systems with an interpreter, it is possible to simply fallback to using interpretation only and implement debugging in the same way as in direct interpretation. (There might be some extra work needed for breakpoints, if the breakpoint is located in the middle of some TU for example.) The case where no interpreter is involved is not covered here.

4.3.8 Summary and conclusion

A jit compiler would probably speed up the execution compared to a pure interpreter, while avoiding many of the problems introduced by “usual” compilation (static compilation, ahead of time). But all that comes with a high implementation cost (compared to a simple interpreter). At planning time, it seemed unlikely that there would be enough time in the project to do jit compilation, and because an interpreter is often used in jit systems it seems better to start writing an interpreter, and if it turns out that there is enough time left after this is done, then start building a jit as well. The alternative involving not using an interpreter at all and starting directly on a jit system risks, in the end of the project, ending up with noting usable at all.

4.4 Short outline of other simulation methods

Hardware supported simulation was not considered in any of the above three methods. Instead, it is discussed in this separate section, but the conclusion is that these hardware supported methods are not appropriate for our situation.

4.4.1 Hardware supported simulation

Hardware supported simulation refers to fully or partly delegating computation to (possibly, custom built) hardware rather than carrying out those computations directly on the computer’s processor. The REPLICA simulator cannot rely on extra external hardware to be available, so hardware supported simulation is neither considered nor discussed in detail here. See Brandner, Horspool, and Krall 2010 for a short discussion on hardware supported simulation.

4.4.2 General-purpose computing on graphics processing units (GPGPU)

Resembling the above proposition, yet not rejectable by the same argument, is general-purpose computing on graphics processing units (GPGPU) – as reliance on GPGPU-capable hardware being available would be an acceptable dependency. At least superficially the PRAM model shares many characteristics with the GPU-part of the programming models used in CUDA and OpenCL, and possibly other GPGPU environments. At least, both PRAMs and GPUs are often thought of as having a high number of cores. However, GPGPU is mainly targeted towards highly data-parallel computations with minimal control flow. With this in mind, mapping (at least, in any direct and straightforward sense) programs written for a PRAM-like programming model to a GPGPU programming model does no longer seem as tempting, because of the high amounts of synchronization (i.e., non-trivial control flow) that seem to be needed because of the PRAM’s strong memory model. A simulator utilizing GPGPU is therefore not considered in more detail than this short section.

21 http://developer.nvidia.com/cuda
22 http://khronos.org/opencl
Chapter 5

Parallel Simulation of Parallel Machines

This far, only simulation of uniprocessor machines has been under consideration. At least for simulation on a high abstraction level, expanding the uniprocessor simulation techniques already discussed to a multiprocessor (shared-memory) machine simulation is fairly straightforward. Simply iterate over the guest processor for each cycle and execute some of the above mentioned simulation technique for each processor (with a weaker memory model than relevant here, it might be possible to execute more than only one cycle before switching to another processor to simulate). How much extra effort is needed beyond this depends on, amongst other, what memory model is assumed in the guest system. For example, an EREW PRAM memory model without error handling is fairly straightforward to implement. Simply use a shared (fixed-size) array to simulate the whole PRAM memory, with proper synchronization between each guest cycle, where each element in the array corresponds to a byte in the PRAM memory.

But, for example, with some CRCW PRAM variant memory model, extra effort is needed to handle the case when two or more processors write to the same memory location in the same cycle. (At least if any write is allowed to collide.) A similar mechanism may be needed to implement error handling for the EREW model. Even worse, cycle count accurately simulating the REPLICA memory model requires even more work, as the shared memory is split up into multiple memory modules that are distributed over an on-chip network. There are therefore more elements to consider here, as messages need to be propagated though the network and so on.

As the guest machine is a parallel machine, the question naturally follows, can the host processor utilize multiple host processors (in an efficient manner) to speed up the simulation? I.e., can we parallelize the simulator? I.e., can we simulate the parallel machine in parallel? Before investigating this, some terms will be introduced.

5.1 Some parallel programming terminology: speedup measures

With minor modifications, all the following definitions, except “scalable algorithms”, are from Keller, Kessler, and Träf[2001]

When investing these questions, we will need to use the terms “speedup” and “scalable algorithm”. In parallel programming the main goal is often to lower the overall program execution time, “relative speedup” and “absolute speedup” are two measure of how successful our attempt at doing this is. “Scalable algorithms” are related to this goal as well, as we will see.

In this section, we assume, without loss of generality, that each processor has one core and only executes one thread. This is because the definitions are borrowed from PRAM theory, where we only consider “processors” and do not bother with “cores” and “threads”. Extending the definitions from e.g. “speedup when using $p$ processors” to “speedup when using $t$ threads” should pose no problem.
Strictly speaking, one should maybe subdivide “speedup” into two different notions of “speedup”, namely “asymptotic speedup” and “empirical speedup”. Given a problem $P$, let $A_p$ denote a parallel algorithm for $P$, and let $A_s$ denote a sequential algorithm for $P$. Furthermore, given an instance of $P$ of size $n$ that we call $P_n$, let $t_{A_p}(n, p)$ denote the worst-case runtime of $A_p$ on $P_n$ using $p$ processors, let $t_{A_p}(n)$ denote the worst-case runtime of $A_p$ on $P_n$ that can be achieved with an unbounded number of processors, and, lastly let $t_{A_s}(n)$ denote the worst-case runtime of $A_s$ on $P_n$. With these definitions we can define two variants of “asymptotic speedup” as follows.

**Definition 3.** The asymptotic absolute speedup of $A_p$ relative to $A_s$ is

$$\frac{t_{A_s}(n)}{t_{A_p}(n)}.$$

The asymptotic relative speedup of $A_p$ using $p$ processors is defined as

$$\frac{t_{A_p}(n, 1)}{t_{A_p}(n, p)}.$$

If we instead were to have empirical measures of the program running time rather than analytically derived expression of asymptotic time complexity the “empirical speedup” measures are of more use. This can be measures in terms of e.g. the number of cycles or wall-clock time. For the same situation as for asymptotic speedup notions, let $t^M_{A_p}(n)$ denote the (measured) running time (in some unit) of (parallel or sequential) algorithm $A$ on machine $M$.

**Definition 4.** In the above context, the empirical absolute speedup of $A_p$ relative to $A_s$ is defined as

$$\frac{t^M_{A_s}(n)}{t^M_{A_p}(n)},$$

and the empirical relative speedup of $A_p$ is defined as

$$\frac{t^M_{A_p}(n, 1)}{t^M_{A_p}(n, p)}.$$

The term “speedup” will be used to refer to any of the above notions of speedup when easily inferred from context which one is referred to.

The running time of the parallel algorithm with one processor $t_{A_p}(n, 1)$ might differ from the sequential algorithm running time $t_{A_s}(n)$ because the parallel algorithm might introduce (for the case with one processor unneeded) synchronization. If we increase the number of processors, we want the speedup to increase as well. If this is the case, we call the algorithm scalable. In the ideal case, we will get e.g. a speedup of 4 when using 4 processors, and a speedup of 600 when using 600 processors. This is not typical, and one cannot expect this to be achieved for all problems, however. Increasing the number of processors will for many algorithms also increase the amount of communication (e.g., synchronization) needed and the speedup will consequently suffer. Moreover, there are other problems that arises when increasing the number of processors. E.g., directly connecting, say, 600 processors to one shared memory module will not result in a speedup of 600 because the memory module cannot handle all the memory access requests without slowdown. As such (non-emulated) shared-memory architectures do not scale (Aho et al. 2007, p. 773), we cannot expect an algorithm implemented on such an architecture to scale beyond a certain limit either.
5.2 PRAM simulation

Consensus is that a direct implementation of the PRAM model is not feasible (Keller, Kessler, and Träff [2001]; Harris [1994]; Forsell [2010]). "A direct implementation" meaning a hardware implementation "structurally similar" to the PRAM model, i.e. a high number of processors directly connected to a shared memory. This is because in real hardware the shared memory module cannot handle an arbitrary number of write and read requests at once as the PRAM model requires, and a bus interconnect cannot handle the amount of traffic needed for processor synchronization (remember, the PRAM model offers strict consistency) and memory accesses. PRAM simulation in hardware is therefore typically, as is the case for REPLICA as well, simulated on a distributed memory machine (Harris [1994]).

When the PRAM model is simulated in software (on desktop computers), because of the memory model offered, synchronization is needed between all processors after each instruction, which is very expensive to implement in software. Consequently, because the extremely fine-grained parallelism (PRAM instructions) and high amount of synchronization needed we can not expect to find a scalable parallel simulation method running on desktop computers – because, indeed, that would be to try directly implement the PRAM in hardware! This is because multicore desktop computers commonly have theirs processors connected to one large shared memory. Efficiently implementing the PRAM model seems to require custom-built hardware, such as the REPLICA architecture.

Nevertheless, some speedup might still be possible, even when using a non-scalable algorithm. If the implementation complexity is not too high this might still be worthwhile – after all, some speedup is better than none at all. In other words, we assume that no scalable algorithm exists and effort will consequently only be spent towards a non-scalable algorithms utilizing a low number of (host) processors.

As mentioned earlier, because of the limitations of shared-memory systems, most attempts of simulating PRAMs (seem) to utilize distributed memory in some way. Harris [1994] breaks the problem of simulating a (CRCW) PRAM on a distributed memory machine into three subproblems, namely:

The concurrent access problem Even if the host machine’s memory modules disallow concurrent access, requests from the guest machine that result in concurrent memory accesses on the host machine must still be serviced correctly.

The memory management problem Mapping the guest address space into the host’s memory modules in such a way that contention is minimized.

The routing / interconnection problem Basically, how are the components of the computer connected and how are messages routed between them?

But as we are mainly interested in PRAM simulation on shared memory, this breakdown of the simulation problem is of limited use here. It does, however, highlight some important problems.

When trying to solve a problem in parallel, it is important to identify independent tasks or where communication is needed between tasks (when they cannot be run independently of each other). For the PRAM variants we are interested in, there might be dependencies between some instructions (i.e., our tasks), so not everything can run in parallel independently of everything else. For example, if two (guest) processors start a multioperation using the same memory location, some form of communication between threads is needed. And writes from all earlier steps must be visible to the current step (this implies synchronization as well). This communication is necessary for not breaking the PRAM programming model and can therefore not be “optimized away”, in any way.

---

1One could do, very complicated, static analysis and instead of assuming the worst-case in every scenario utilize the information gained from the analysis to remove some of the synchronization points that would be used when assuming the worse case, and thereby hopefully speed up the execution. This, however, is beyond the scope of this thesis.
5.2 PRAM simulation

5.2.1 The phase algorithm

The PRAM simulation discussion here will be mainly focused on, what the author has chosen to call, “the phase algorithm.” This name is used because each PRAM step is split up into two phases, one read phase and one write phase. In the read phase, pre-memory ALU and memory read operations are executed. In the write phase, memory write operations and the rest of the operations are executed. All phases are separated by a barrier, as displayed in listing 5.1.

```c
read_phase();
barrier(); // barrier 1
write_phase();
barrier(); // barrier 2
```

Listing 5.1: Pseudocode for the phase algorithm.

Consider step $s$, and let $x_s$ denote phase $x$ for step $s$. Barrier 1 is needed because `write_phase` cannot be executed before `read_phase` is completed, because `read_phase` should only see the values from the previous step $s-1$. Barrier 2 is needed because `write_phase` needs to be completed before `read_phase_{s+1}` is executed. Conversely, it should be obvious that no two phases can overlap.

Each REPLICA thread will execute the phase algorithm (i.e., the code in listing 5.1). When synchronizing for barrier 2, control-flow information is calculated as well. Because, e.g., if some thread hits a breakpoint, all other threads need to be informed of that (and stop execution as well, so control can be returned to the user). Furthermore, the simulator as a whole needs information whether there still is some thread running. Let $T$ be the set of all REPLICA threads, then the following information is calculated each time barrier 2 is executed:

$$B = \bigvee_{t \in T} \text{hit_breakpoint}(t) \text{ and } H = \bigwedge_{t \in T} \text{halted}(t),$$

where $B$ iff some thread hit a breakpoint, and $H$ iff all threads have halted. A thread halts by executing the (pseudo-)subinstruction `HALT`. But depending on how one interprets the subinstruction, it might be possible to save some execution time. If one uses $H$ as a stop criterion, then all threads have to call `HALT` before the machine stops. Alternatively, if one uses the halting criterion that the machine should halt after the first `HALT` execution, it might be possible to save some time. Because one can then piggy-back this on the breakpoint criterion (because both then would take the logical or of all values).

Multioperations and multiprefix operations fit into this algorithm as well. Extra locks are needed to ensure that no writes are lost. Using e.g. atomic variables (such as `std::atomic<int32_t>`) introduces problems regarding how to synchronize the atomic variable with the memory location without introducing extensive overhead. How many concurrent (meaning, occurring in the same step) multioperations are allowed depends on the number of memory modules, and on what addresses these concurrent operations are allowed depends on how memory addresses are mapped to memory modules. In instruction accurate simulation however the memory modules are not explicitly modelled (and therefore not the translation from memory address to memory modules), so one would have to use some kind of stub-like translation to allow for different threads to take different locks depending on the memory location addressed. So, in the instruction accurate case, there will be some function `std::mutex& get_mutex_that_is_used_for_memory_location(addr)` or something similar. This mutex is then held for the entire duration of the multioperation for the executing thread. It some other thread would write to the same location using e.g. the subinstruction `ST` the resulting value is undefined. If we wanted

---

2 Note, mainly by accident, the phase algorithm implements a stronger memory model than REPLICA’s. This does not affect the programmer, the program execution is still correct, but using a weaker memory model would allow for faster simulation.

3 This is mainly a placeholder, because the old convention that used the `TRAP` subinstruction will probably be replaced in the future.

4 Because the latter interpretation allows for less overhead, this was used in the actual simulator. A simple test showed that the speed difference was actually significant.

5 That this is a problem might not be apparent from this short description, however.
some meaningful value for this situation as well, we would have to check for write collisions, which would be very expensive.

Recall the notation introduced when discussing multioperations and multiprefix operations. Subinstructions from the \( M(\mathcal{P}) \) \( z \) group writes directly to memory, while subinstructions from the \( BM(\mathcal{P}) \) \( z \) group first writes to a guest processor-local structure corresponding to a scratchpad, and subinstructions from the group \( EM(\mathcal{P}) \) \( z \) writes the result of the \( BM(\mathcal{P}) \) \( z \) subinstructions to shared memory. I.e., the implementation corresponds fairly directly to how the multioperations and multiprefix operations are implemented in REPLICA. Step caches are not used because the overhead of simulating them in software. The overhead of simulating them would (probably) be greater than the time saved from not writing to shared memory, because the expected case is that at for each memory location at most one thread will write to that location. Reads are not as expensive as writes.

5.3 Mapping guest threads to host processors

We have yet to address the problem that the number of host processors is expected to be smaller than the number of guest threads. Using one host thread per guest thread will therefore not be an efficient usage of the host machine. This is solved by mapping one or more guest threads to each host processor. By mapping is meant to assign some subset of the guest threads to each host processors, the mapping being either statically or dynamically calculated (i.e., and might or might not change during runtime). This is not problem specific to REPLICA simulation, and the reader might recognize many of the mappings from e.g. OpenMP

\[ \text{http://openmp.org} \]

The phase algorithm (listing 5.1) must be modified when not using one host thread per guest thread. When using e.g. a static mapping the modified algorithm in listing 5.2 could be used.

```
for (auto & thread : threads_scheduled_for_this_processor)
    thread.execute_read_phase();
barrier(); // barrier 1

for (auto & thread : threads_scheduled_for_this_processor)
    thread.execute_write_phase();
barrier(); // barrier 2
```

**Listing 5.2:** Pseudocode for the phase algorithm when not using one host thread per guest thread.

5.3.1 Static processor mappings

As stated in the beginning of this section, one possible solution is to map each guest thread to an own host thread in the simulator. But as the expected case is that the number of guest threads will be much larger than the number of host processors, this mapping will introduce extensive context-switching overhead. This mapping is shown in figure 5.1 where \( G_i \) denotes guest thread \( i \), \( T_i \) denotes host thread \( i \) and \( H_i \) denotes host processor \( i \). Henceforth, because of the overhead, we will assume that the number of host threads used equals the number of host processors, and therefore we will only talk about host processors rather than both host threads and host processors.

Reducing the number of host threads, to e.g. the number of host processors available will improve the situation somewhat. The amount of context-switching is lowered because more than one guest thread is mapped to one host processor. An example of such a static mapping is, say, with 11 guest threads and 4 host processors, that host processor 0 takes guest thread 0, 1 and 2, host processor 1 takes guest thread 3, 4 and 5, and so on – as illustrated in figure 5.2.
5.3 Mapping guest threads to host processors

Consider each guest thread as a task. If we would have a set, of statically known cardinality, of independent tasks and each task requires the same amount of work, a static mapping is the best possible mapping as it imposes no extra runtime overhead (compared to dynamic mappings) and we get the best possible division of work between processors. The problem is that all these assumptions does not necessarily hold for a set of guest threads. For example, different threads might in some steps execute different instructions (which might happen even if all threads execute the same program\textsuperscript{7}) that each require a different amount of work. As all host processors have to wait for the slowest processor to finish (the slowest being the processor that happens to get the most amount of work in that step), some processors will just idle rather than carrying out some meaningful work (e.g., somehow helping the processors that has not finished yet).

Given a static mapping, it is often easy to write a program designed to “break” this mapping, in the sense that the execution will be inefficient. If the simulated program splits the work unevenly between guest threads, as previously stated, the above mapping will not result in an efficient execution. Consider for example listing 5.3.

With the host computer from above, $H_0$ and $H_1$ will have to wait for $H_2$ and $H_3$ because they will take longer time to complete theirs execution paths.

\begin{verbatim}
int main() {
    if (id < get_number_of_processors() / 2) {
        small_amount_of_work();
    } else {
        large_amount_of_work();
    }
    barrier();
    if (id == 0)
        display_the_result();
}
\end{verbatim}

\textbf{Listing 5.3:} Example program where the work is unevenly split between guest processors.

It is however possible to solve this specific situation by using another mapping, e.g. as in figure 5.3.

As said before, regardless of how a static mapping is constructed, there will always be some way of writing a guest program that results in poor utilization of the host machine (regardless of whether it is on purpose or by accident). The common theme among these programs that break static mappings is that they split work unevenly between guest threads, one could therefore argue that they are badly written and would execute inefficiently on a

\footnote{Control flow might for example depend on the thread’s id, or the result of some pseudorandom number generator.}
real REPLICA machine as well. I.e., that it is the programmer’s responsibility to evenly split the work between guest threads and not the simulator’s. Before a final decision is made, we will look at dynamic mappings.

5.3.2 Dynamic processor mappings

As a possible solution to the problems of static mapping, dynamic mappings will be considered as well. One fairly simple and naive, but straightforward, example of a dynamic mapping is as follows. In the beginning of each step put a tuple (absolute thread id, instruction) on a shared queue. The available host processors then proceed by executing, one by one, each task (i.e., a tuple here) from the queue until the queue becomes empty. Because all threads contend for one shared resource, namely the queue, this will probably not scale very well. But, in a program with unevenly spread out work between threads, the better load balance gained from using a queue would hopefully outweigh the overhead of sharing the queue between processors. This dynamic mapping is illustrated in figure 5.4.

As for example in listing 5.3, even if the threads with lower ids finish before those with higher ids the host processors that first executed the guest threads with lower ids will simply proceed with executing the guest thread with higher ids.

Dynamic mappings, unlike static mappings, introduce runtime overhead. There are obviously multiple ways to implement the shared queue, e.g. using one lock for the whole queue (very simple, but probably slow), or even some non-blocking implementation (which is more difficult to implement, but hopefully faster). There are also multiple strategies on how to fetch tasks from the queue, e.g. one task at a time, or some fixed number of tasks, or start out by taking large blocks of tasks and then a smaller and smaller number each time. Regardless of how these aspects are implemented, there will still be more overhead than a static mapping for well-balanced programs.

5.3.3 Efficient mappings when also considering multioperations and multiprefix operations

When mapping guest threads to host processors, we have not considered if the two threads mapped to one host processor belong to the same guest processor. When using instruction-accurate simulation this does not matter in most cases. But as we know from previously in this chapter (and previous chapters as well, such as section 3.2.2 more specifically), when executing multioperations and multiprefix operations some guest processor-global operations are carried out, i.e. writing to the processor-associated scratchpad. If guest threads from different guest processors are mapped to host processors without considering what guest processor each guest thread belongs
5.3 Mapping guest threads to host processors

to, synchronization might be needed when accessing the scratchpads (and other processor-local resources). If all guest threads that belong to a guest processor are mapped to the same host processor, we know that at most one host processor will access each scratchpad, and consequently we do not need any synchronization. Therefore, it is better (with respect to the amount of synchronization needed) to map whole guest processors to host processors, instead of individual threads. In a sense, we are still mapping guest threads to host processors, but with the restriction that all guest threads from one guest processor always end up on the same host processor.

As this is an additional constraint, this might introduce load imbalance in other cases (not related to multioperations or multiprefix operations). Consider the case where we have 3 host processors, 4 guest processors and 10 guest threads per guest processor. That is, in total 40 guest threads. When mapping guest threads to host processors we could use the assignment in table 5.1.

<table>
<thead>
<tr>
<th>Processor</th>
<th>Number of threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>14</td>
</tr>
<tr>
<td>2</td>
<td>13</td>
</tr>
<tr>
<td>3</td>
<td>13</td>
</tr>
</tbody>
</table>

Table 5.1: Number of guest threads per processor, when not considering which guest processor each guest thread belongs to.

But this mapping is not possible if all guest threads that belong to one host processor must belong to the same guest processor. With this extra constraint, we would have to use something like table 5.2.

<table>
<thead>
<tr>
<th>Processor</th>
<th>Number of threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>20</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>3</td>
<td>10</td>
</tr>
</tbody>
</table>

Table 5.2: Number of guest threads per processor, when considering what guest processor each guest thread belongs to.

One additional advantage of using this kind of mapping, and static mappings in general, is that host caches might be more efficiently used compared to dynamic mappings. If guest thread $t_0$ accesses variable $v$ we expect $t_0$, by the principle of locality, to access it soon again. If $t_0$ is executed on host processor $h_0$ when the first access to $v$ occurs, we get a cache miss. If at a later time $t_0$ accesses $v$ again, but is now executed on host processor $h_1$ we might get another cache miss. The exact number of cache misses depends on host hardware and the guest program. For example, if $h_1$ executed another guest thread $t_1$ that also accessed $v$ recently, we do not get any cache miss when $t_0$ tries to access the same variable the second time (when executed on $h_1$). Or, for example, if $h_0$ and $h_1$ share L2 cache, $h_1$’s access to $v$ might give us a L1 miss but instead of having to go to memory we find it in the shared L2 cache.

The above holds for any static mapping, but the question is if mapping guest threads from the same guest processors to the same host processor improves this further. This would be the case if, e.g., guest threads from the same guest processor accessed more of the same memory compared to two guest threads from two different guest processors. Whether this is true or not is unclear. One reason not believing so is that the REPLICA programming model is mainly focused on threads rather than processors, and more specifically not on what guest processor each guest thread is executed on. So the programmer will probably not (explicitly) treat threads from two processors any different from threads from the same processor (this is needed for efficient use of multioperations and multiprefix operations utilizing scratchpads however).

The above is true if $v$ is a guest register as well. But depending on the number of guest threads per host processor and host hardware this might or might not make any difference. If the cache replacement strategy used is e.g. last
recently used and the number of guest threads are so many that all their registers cannot fit in host cache at the same time, the guest thread’s register contents might be swapped out between executions.

5.4 Related work

Almer et al. [2011] presents an LLVM-based parallel tracing jit functional level simulator for an embedded multicore platform implementing the ARCompact ISA. Compilation and simulation are carried out in parallel with the help of a parallel task farm for compilation, and each guest processor is mapped to one host thread, where the host OS is used for scheduling if more guest processors than host processors are simulated. A 32-core x86 machine is used to simulate 2048 guest cores, and for a low number of guest cores a speedup compared to sequential simulation is achieved.

Wesarg et al. [2007] and Keller, Kessler, and Wesarg [2009] attempt to parallelize the SB-PRAM simulator pramsim (Keller, Kessler, and Träff [2001]), although with moderate success (Brenner, Keller, and Kessler [2012]). Keller, Kessler, and Wesarg [2009] exploits that so-called Fork groups can be executed independently of each other and because of the approach used, even without utilizing multiple host threads some speedup is gained because busy waiting when groups are joined is avoided. But, because we are not compiling a high-level language here we cannot rely on the presence of this group structure and the approach does therefore not directly apply here. Wesarg et al. [2007] combines time warp (optimistic) parallel discrete event simulation and shared memory programming techniques to speedup the SB-PRAM simulation. This is of more relevance here, as this approach does not depend on any Fork constructs. But because lack of time, this was not investigated further. Lastly, Brenner, Keller, and Kessler [2012] investigates the possibility of simulating PRAMs on GPUs (as previously mentioned in section 4.4.2), more precisely Fork simulation using CUDA.

5.5 Conclusion

For well-balanced guest programs a static mapping is probably the best choice (because a static mapping imposes no additional scheduling overhead), but imbalanced guest programs would probably profit from using a dynamic mapping (with dynamic load balancing). The best possible solution would be that both well-written and poorly written guest programs would be executed efficiently and the only difference the user would observe would be some (maybe significant) increase in the number of (REPLICA) cycles needed to execute the imbalanced program compared to the well-written one.

This, however, is an unrealistic expectation. Explicitly, the situation is as follows. We are given an imbalanced program and expect the simulator to execute it as efficiently as a well-written program. That is, we are asking the simulator to automatically parallelize the program – which is, if even possible, beyond the scope of this thesis. (This argument is developed further in the final conclusion of this thesis, i.e. chapter 8.) The dynamic mapping described above might improve some programs, but not all. Therefore, a simple static mapping will be used in the simulator.

As an example of the above, consider the mapping of guest threads to host processors. If the guest program utilizes many (e.g.) multiprefix operations, using the additional constraint presented in section 5.3.3 is probably going to improve throughput as less synchronization is needed compared to allowing threads from the same guest processor be mapped to different host processors. But in a program not using multioperations or multiprefix operations at all, and we get the situation presented in table 5.1 and 5.2, we probably want to not use the mapping constraint because we would get better load balance without it (assuming the guest program is balanced as well), and not gaining anything by having threads that belong to the same guest processor belong to the same host processor.

8For example, in the Fork language, branches originating from a control-flow split that depended on thread-local data (such as thread id) are not guaranteed to execute in lockstep.
as well. The same is true for the step caches. It is easy to imagine guest programs that would improve from the presence of step caches (multiple threads writing to the same memory location in the same step frequently). It is also easy to imagine guest programs that would only see step caches as unnecessary overhead (programs where for each memory location at most one thread write to it). This can be summarized as that we cannot abstract away the underlying host machine completely.

As some decision has to be made and as we do not want to introduce more synchronization overhead than needed, we will map all threads that belong to a guest processor to the same host processor. With the expected configuration parameters it should be possible to still have a fairly balanced workload.

Regarding the amount of synchronization needed (when using a static mapping, such as listing 5.2) to simulate the REPLICA machine depends on the machine parameters and the guest program. For the PRAM model, synchronization is needed between all instructions. For REPLICA the situation is somewhat better, as we only need to synchronize twice for each instruction, and not once per subinstruction (that loosely corresponds to one PRAM instruction). So if we have long instructions (i.e., many subinstructions per instruction) the time between synchronization becomes larger. And the more guest threads, the more iterations each guest thread will do in each for loop (again, in listing 5.2), and consequently the longer between synchronization points.

In summary, how efficient the simulation will be depend on both the guest program and the guest machine’s parameters, but it is (seemingly) difficult to reduce the overhead involved in simulating REPLICA machines. For e.g. the situation where the instructions are short and a low number of guest threads are used, using just one host thread (and therefore being able to skip synchronization entirely) might be more efficient than utilizing all host processors.
Chapter 6

The Design and Implementation of the REPLICA Simulator

The overall design of the simulator is fairly straightforward. The simulator begins program execution with parsing, using ANTLR (Parr 2007), the assembly program specified by the user and builds an internal representation (IR) of the program. The entities in the internal representation correspond directly to the textual representation of the REPLICA input program. At a high level, the IR is a list of instructions, each consisting of instruction-wide operands (OPs) and subinstructions, the latter in turn consisting of an operator and one or more operands. The IR and its components are mainly built from the container classes available from the C++ standard library. Some more, standard and uninteresting, tasks are handled during parsing, such as building a symbol table and deciding where to allocate variables. Such kind of straightforward and standard tasks are not described in this thesis, this holds true for all parts of the simulator.

There are three important class hierarchies called “simulation execution model”, “processor” and “memory”. For instruction accurate simulation this is not really needed, because the simulation execution model mainly calls the processor class hierarchy directly, without doing any useful work itself. For instruction accurate simulation, there is just one instance of the processor hierarchy and one instance of the memory hierarchy. The processor instance keeps track of all threads, and the memory instance is essentially a PRAM memory module. For cycle count accurate simulation, the idea was that there would be one instance of the processor hierarchy per processor module, and the same for the memory hierarchy. There would have also been a network class hierarchy, with switches, packets and so on. An instance of the simulation execution model would then be used to “drive” the simulation forward (cycle by cycle), e.g. each memory, processor and network instance could have a member function called cycle that would simulate one cycle of execution on the relevant module. But for instruction accurate simulation, the simulation execution model becomes trivial, as said above. All this is illustrated in figure 6.1.

For actually executing instructions on the simulated machine, a token-threaded interpreter is used (i.e., very much like a switch-based interpreter) in combination with the phase algorithm (listing 5.1).

There is also a simple user interface for running programs and displaying the symbol table and variable contents. For example, typing r would run the program, and typing var foo would display the value of the shared variable foo.

---

1 Which was initially planned, but later removed because of time constraints.
6.1 Some design choices

The following sections contain some decisions that were made, and shorter discussions on why they were made.

The data structures used for program execution are, as said, straightforward. Because the simulation is only instruction accurate, and therefore the network is not modelled, REPLICA's shared memory is essentially modelled as a long array that can be accessed directly. Thread and processor state (such as registers and scratchpad results, respectively) are stored in structs, with e.g. an array of integers for the general purpose registers. Overall program design has much in common with the example interpreter from listing 4.3.

### 6.1.1 Expressing time progression in a parallel system as serial computation

Consider figure 6.2 If we only care about the input–output relation (namely, $o_0 = i_0 \land i_1 \land i_2$) it does not matter whether we are using the circuit of figure 6.2a or 6.2b. But at a more detailed level, there are indeed differences between the referred figures. For example, as there are more gates (in the longest path) between the inputs and the output in the first circuit, changes in the first circuit might take longer to propagate to output compared to the second circuit. If $C_0$, $C_1$ and $C$ for some reason were synchronous gates sharing a common clock, the first circuit would need two cycles to update the output, while the second circuit only would need one. These kinds of timing issues arise when trying to express time progression in a parallel system as serial computation. If the internal state (i.e., the “stuff” between the inputs and the outputs) has to be modelled, extra care has to be taken to these kinds of timing issues.

In instruction accurate simulation it is simple to execute all components in an order that does not break any
dependencies. The above short discussion is mainly relevant for cycle count accurate simulation, and because this was not implemented it is not discussed further. However, in REPLICA's network, connecting the processors with the memory modules, these kinds of timing issues arise when sending packets between nodes in the network. One therefore has to be careful of in what order different components are progressed from cycle \( c \) to cycle \( c + 1 \).

### 6.1.2 Some problems associated with using multiple labels for the same instruction or data

There are essentially two independent queues for handling data and instruction labels. Consider listing 6.1.

```
L0:
L1
L2: .WORD 0xFF
    OP0 1 ADD0 R0,1 WB0 A0
```

**Listing 6.1:** Label example 1.

In this example, the labels \( L0 \) and \( L2 \) will refer to `.WORD 0xFF`, and the label \( L1 \) will refer to \( OP0 1 ADD0 R0,1 WB0 A0 \). So the queues work independently of each other, this is an implementation detail and not the only possible alternative. One could, for example, say that the above program is misformed and instead signal an error.

There is also a somewhat related problem, which is a consequence of how data is specified to be either shared or local. Consider listing 6.2.

```
L0_:
L1: .WORD 0xEE
L2: .WORD 0xDD
```

**Listing 6.2:** Label example 2.

This example, on the other hand, is misformed. All labels for a memory location must specify that the associated variable be either local or shared, any combination of the two is not allowed. That is, all labels must agree on whether the variable is shared or local. Allowing such combination would only lead to confusion, as a label that has the name of a shared variable still could refer to a local variable if for example we instead adopted the convention that the last label decides whether the data is shared or local (e.g., using this convention, in listing 6.2 \( L0_\) would refer to a local variable).

### 6.1.3 Breakpoint handling

There are (as always) multiple ways to implement debugging breakpoints. One simple way is to have a hash table (e.g., `std::unordered_set`) and compare the thread’s program counter each time a new instruction is
executed. This, obviously, imposes overhead, even when breakpoints are not used. An alternative solution, that does not introduce any overhead when breakpoints are not used, is to insert the (pseudo-)subinstruction \texttt{BREAK} in the IR that breaks program execution when executed. In this way, no extra checking is needed, but some extra work is needed to modify the IR and to keep track of the breakpoints. This latter solution was used in the actual simulator.

Usually, when one adds a breakpoint to (say) line 5 one would expect the program to stop when line 5 is reached. Because this would be expensive, even when breakpoints are not used, to implement, this is not the convention used in the REPLICA simulator. Instead if we would have a breakpoint on line 5 the program would break \emph{after} line 5 was executed. This allows for a more efficient breakpoint implementation.

\subsection*{6.1.4 The rationale behind how variables are displayed to the user}

Given a sequence of bits, one cannot (reliably) interpret them without also being given the type associated with the bits (e.g., \texttt{unsigned short int} or \texttt{signed long int}). Therefore, only data memory with a specified type can be displayed in the simulator. Furthermore, to refer to a memory location, some kind of name must be used. This could be e.g. an address or a (symbolic) name. For simplicity, only memory locations with names and a type specified can be displayed in the simulator.\footnote{Actually, the signedness of the specified type it is not known from the information that can be extracted from the program, and all types are therefore assumed to be signed. Obviously, this is an incorrect assumption. But no better (less broken), the author claims, assumption can be made.}

To display a shared variable, one simply enters the name of the variable. To display a thread-local variable, one has to provide the name of the variable and the processor and thread number as well.
Chapter 7

Results and Evaluation

As concluded in previous chapters, because the simulator is a fairly thin layer between the host machine and the REPLICA programming model, the host machine’s cost model “leaks through” fairly directly. For example, memory access patterns with neither temporal nor spatial locality will be more costly than access patterns with these properties. Of course, for a “pure” PRAM there should not be any difference between these cases. Note that if acquiring and releasing locks flushes caches and write buffers too aggressively, accesses from different steps might not affect each other to any large extent. For example, on the test machine later specified iterating over an array in a native single-threaded program such that we get one cache miss per iteration takes twice the time of iterating over the same array but accessing element after element in increasing order with stride size one, resulting in much fewer cache misses. Doing the same thing in a REPLICA program, with 400 threads, the two program versions runs in exactly the same amount of time.

As said in the concluding section of the chapter on parallel simulation (section 5.5), the synchronization overhead is expected to be significant, which figure 7.1 corroborates. The program used to generate the graph would in C++ corresponds to listing 7.1.

```cpp
int main() {
    for (int i = 0; i < 1000; ++i)
        ;
}
```

Listing 7.1: Example program 1, code used to generate figure 7.1 i is allocated into a register. Using 10000 iterations instead gives the same measured results.

Removing the barrier synchronization (i.e., the two barriers from the phase algorithm, from listing 5.1) results in significant speedup. (The transformation is, obviously, not valid in general, because all threads must execute in lockstep.) The machine used in the measurements has an Intel Atom processor (D525), which has two hyper-threaded cores (i.e., four in total). We can see that (probably because synchronization overhead) two host threads allows for faster execution than utilizing all cores.

As seen from the figure, when the number of threads increases, the number of instructions per millisecond decreases. We think that this is because for a small number of guest threads the guest registers can fit in host cache, while using a larger number of threads some reads have to go to main memory instead. That is, we are using host main memory to simulate guest registers. Figure 7.2 is consistent with this conclusion because the number of instructions per millisecond stops decreasing after approximately 10000 threads.
Figure 7.1: Measured synchronization overhead, the program just loops 10000 times without doing anything (listing 7.1).
Figure 7.2: Using host main memory to simulate guest registers does not result in any impressive simulation speed. Still using the program from listing 7.1.
We also measured results for multioperations, namely by executing the programs in listings 7.2 and 7.7. First, we tried some different assembly representations (listings 7.3, 7.4, 7.5 and 7.6) of the first program, listing 7.2, with results presented in figures 7.3, 7.4, 7.5 and 7.6, respectively.

These tests were meant to illustrate the synchronization overhead imposed by splitting up instructions into multiple instructions. For example, listing 7.3 has 4 barriers per loop iteration, while listing 7.6 will need 8 barriers per iteration. However, the result, as seen from the graphs are somewhat surprising because the programs with more barriers are faster (more instructions per millisecond) than the programs with fewer barriers. What causes this remains unclear. This is assumed to be an anomaly that does not hold in general.

```c
int sum_ = 0;
// Each thread executes main
int main() {
    for (int i = 0; i < 1000; ++i) {
        MADD thread_id, sum_;
    }
}
```

**Listing 7.2: Example program 2**

```assembly
.ProgramStart OP0 __thread_id ADD0 R32, O0 LD0 A0 WB0 M0
Loop OP0 sum_ OP1 1 ADD0 O1, R1 MADD0 R0, O0 WB1 A0
    OP0 1000 OP1 Loop SEQ O0, R1 BEQZ O1
HALT
sum_:_ .WORD 0x00
```

**Listing 7.3: One possible assembly version of listing 7.2, where ADD is before MADD.**

**Figure 7.3: Results for listing 7.3**
Listing 7.4: Another possible assembly version of listing 7.2 where ADD is after MADD.

Figure 7.4: Results for listing 7.4
Listing 7.5: Another possible assembly version of listing 7.2, where ADD and MADD are in separate instructions.

Figure 7.5: Results for listing 7.5
Listing 7.6: Another possible assembly version of listing 7.2 where even more instructions are split up into more instructions.

Figure 7.6: Results for listing 7.6
Lastly, listing 7.7 computes the same sum, but using multioperations utilizing scratchpads. Results are presented in figure 7.7.

```c
int sum_ = 0;
int main() {
    for (int i = 0; i < 1000; ++i) {
        BMADD thread_id, sum_;
        EMADD thread_id, sum_;
    }
}
```

**Listing 7.7: Example program 3**

```assembly
.ProgramStart OP0 _thread_id ADD0 R32, O0 LD0 A0 WB0 M0
Loop OP0 sum_ BMADD0 R0, O0
    OP0 sum_ OP1 1 EMADD0 R0, O0 ADD0 O1, R1 WB1 A0
    OP0 1000 OP1 Loop SEQ O0, R1 BEQZ O1

HALT

sum_: .WORD 0x00
```

**Listing 7.8: A possible assembly version of listing 7.7**

![Figure 7.7: Results for listing 7.8](image-url)

**Figure 7.7: Results for listing 7.8**
Chapter 7. Results and Evaluation

7.1 Comparison to other simulators

The “new” REPLICA simulator is much faster than the “old” REPLICA simulator. For example, the old simulator, on an Intel Core 2 Duo (MacBook Pro 7.1) listing 7.3 takes 3 minutes and 15 seconds to run with 64 processors and 512 threads per processor. With the same configuration, but using the new simulator on the Intel Atom computer, the simulation completes after 15 seconds (using one host thread). Running the old simulator, with the same configuration, but with listing 7.7 instead takes 5 minutes and 5 seconds. Using the new simulator we instead get the simulation results after 14 seconds when using four host threads, and 16 seconds when using 1 host thread. When running the same program but with only 4 processors (and still 512 threads per processor) the simulation takes 20 seconds to complete with the old simulator, using the new simulator the simulation ends under 1 second (both when using one and four host threads).

The simulator was compared against the SB-PRAM simulator pramsim (Keller, Kessler, and Träff 2001) as well. Listing 7.9 was used and corresponds to the REPLICA assembly program from listing 7.8. The actual assembly program used in the simulator is located in appendix B. Using 128 physical processors, with 32 virtual processors each, the simulation takes 25 seconds on the Intel Atom machine from above. With 128 processors with 32 threads each, using two host threads, the REPLICA simulator runs listing 7.7 in 1.5 seconds. When using only one host thread the execution takes 2 seconds instead.

```sh
int sum_ = 0;
// Each thread executes main
void main( void ) {
    int res, i;
    for (i = 0; i < 1000; ++i)
        res = mpadd(&sum_, $);
}
```

Listing 7.9: A Fork version of listing 7.7

The REPLICA simulator is not, however, very fast compared to native execution. Executing the C++ program in listing 7.10 natively on the host machine (again, the Intel Atom machine) takes less than 1 second – around 0.6 seconds to be more precise. Using only one processor with one thread, and changing the number of iterations to 200000000, listing 7.3 takes 2 minutes and 54 seconds when run in the simulator. This comparison is somewhat unfair because the simulator is built for running a high number of threads. So when using only one thread the simulator executed only 2000 instructions per millisecond. We know from the earlier graphs that this number is higher when using a higher number of threads.

```cpp
#include <iostream>

int main() {
    int sum_;
    for (int i = 0; i < 200000000; ++i)
        sum_ += 1;
    // So that sum_ is not optimized away.
    std::cout << "Hello there, the result is " << sum_ << std::endl;
}
```

Listing 7.10: Simple program to compare native versus simulation speed.

7.2 Some future work

There are, as always, some minor (arguably not very interesting) possibilities for future work. E.g., better error reporting for assembly syntax errors, and better reporting of out of bound accesses when the guest program does
something illegal. A better, more well-documented, user interface is needed as well (especially for debugging purposes). A more important task is to fix the backward-compatibility problems with the old simulator that still exist, and that results in that programs written for the old simulator cannot run on the new simulator (without modifications). The details that are left seem to require reverse engineering the old simulator and extrapolating information from existing programs.

Some more substantial tasks include: The current simulator does not in any way try to execute inefficient programs efficiently. There might be some important special cases from this class of programs that the simulator can take into consideration and in some way execute efficiently even though the program would not execute efficiently on a REPLICA machine. Another task is to look into ways of minimizing the synchronization overhead, such that using multiple threads yield actual speedups rather than slowdowns. Another task is to model all details of the REPLICA machine that are required for cycle count accurate simulation.
Chapter 8

Conclusion

This chapter gives a short note on why one should probably not be particularly surprised by the fact that it seems difficult to write a fast and efficient REPLICA simulator.

Forget, for a short moment, that we are writing a machine simulator. If one were given the task of writing a virtual machine for language $X$, that supports multithreading, it seems like an unreasonable expectation that when the virtual machine is given a badly written/inefficient program (written in $X$, of course), that it should still be able to execute it efficiently. It seems that this is for example not expected from implementations of mainstream languages such as Java or Python.

With this in mind, can will still, under any meaningful interpretation, claim to have built a “fast and efficient” REPLICA simulator (that was, although maybe not explicitly said in chapter 2, one of the original goals). Because efficiently executing inefficient programs are not expected from other virtual machines, let us solely focus on efficient programs. By an efficient program is meant a program that would execute efficiently on a real REPLICA machine. The hope is then that this would relieve us from having to transform/optimize the program from the given inefficient form to an efficient program before it is executed (or even during execution, if one consider profile-guided optimization as well).

The problem is that this restriction is not enough either. One cannot simply execute the program (in any direct manner) because the host machine and the guest machine are so different, especially with respect to each machine’s cost model. Consider all steps one usually have to go through when designing parallel algorithms for non-PRAM machines (Foster 1995), such as minimizing communication cost, how to divide the program into tasks, and how to map these tasks onto actual processors and so on. Now consider instead all promises that PRAM advocates (e.g., Forsell 2010) sets out about how much easier it is to program a PRAM than a (non-PRAM) usual desktop computer, because we for example do not have to struggle as much with minimizing communication costs, partitioning the problem is easier and so on. Still, for a program to be efficiently executed on a desktop computer these aspects have to be taken into consideration. Because the programmer who wrote the REPLICA program did not have to take this into consideration, and the only “intermediary” between the REPLICA program and the desktop computer is the simulator, the responsibility of doing all the steps usually required from programmers of parallel programs (again, e.g., minimizing communication and synchronization) falls on the simulator. Therefore, the problem is an instance of automatic parallelization. Because of the limited success of such endeavors in other contexts (especially for highly non-regular problems), we cannot expect a REPLICA simulator to efficiently execute all program that would execute efficiently on a real REPLICA machine.

In summary, one cannot program efficient parallel programs when ignoring hardware details (Herlihy and Shavit 2003), such as memory hierarchies. But using the REPLICA programming model, these aspects (hardware details) are not visible. Therefore, much of (arguably, the most difficult) work usually done by the programmer when programming directly for the host machine is instead put on the simulator.
This concludes the thesis. The above can be summarized as *that it seems that* neither inefficient nor efficient REPLICA programs can (in general) be executed efficiently on desktop machines without involving non-trivial transformations on the given program. This regardless of whether one interprets efficient as actually efficient or without overhead imposed by the simulator\(^1\).

\(^1\)We do not in any way claim that the above is an absolute proof of these statements. We acknowledge that this whole last section is highly speculative. Maybe a better formulation would be “these are some problems that we did not manage to solve, and furthermore these problems seem difficult”.
Bibliography


Appendix A

Simple Interpreter Implementation Details

```cpp
#include <iostream>
#include <cstdint>

struct registers_t {
    // General purpose registers
    int32_t r0, r1, r2;

    // Condition register, the result of // condition instructions are stored here.
    bool cond;

    // Program pointer, points to // next instruction to be executed.
    std::size_t pc;

    // True iff the machine is halted.
    bool halted;
};

enum instruction_op_t {
    ADD_OP,
    SET_OP,
    CMP_EQ_OP,
    DISPLAY_OP,
    JUMP_OP,
    JUMP_IF_TRUE_OP,
    HALT_OP
};

struct instruction_t {
    instruction_op_t op;
    int32_t *r0, *r1, *r2;
    std::size_t addr;
    int32_t val;
};

void boot_machine(registers_t& regs) {
    regs.pc = 0;
    regs.halted = false;
}
```

// Will return true iff the machine is not halted,
bool get_next_instruction(const instruction_t* program,  
const registers_t& regs,  
instruction_t& inst) {
    if (regs.halted)
        return false;
    inst = program[regs.pc];
    return true;
}

void print_halt_message(const registers_t& regs) {
    std::cout << "The machine has halted, its final state being:
" << "r0 = " << regs.r0 <<", r1 = " << regs.r1
    << ", r2 = " << regs.r2 <<\n" << "cond = " << regs.cond <<", pc = " << regs.pc
    << std::endl;
}

Listing A.1: Implementation details for listing 4.3
Compiled Fork Program

```assembly
#include "forkaliases"

.section ".gsdata", .data
.section ".text", .text
jra forklib_startup

.section ".gsdata", .data
.globl _sum_
_sum_: .long 0

.section ".text", .text
.globl _main
_main:
// async void main(pr void), shfrmsz=1, prfrmsz=2, ncalls=0
psheq .fpp,.spp /*save old fpp*/
ldg .fpp,0,.Ret /*get caller’s $$*/
mov .spp,.fpp /*set new fpp*/
stg .Ret,.fpp,0 /*set $$*/
add .fpp,3,.spp /*alloc space for locals*/
add .fpp,1,.r18 /*ADDRLP*/
gtlo 0,.r19 /*CNSTI*/
stg .r19,.r19,0 /*ASGNx*/
L1:
gthi _sum_,.r18 /*ADDRSGP*/
add .r18,(_sum_)&0x1fff,.r18 /*dto*/
ldgn .r19,0,.r19 /*INDIRx*/
mpadd .r18,0,.r19 /*MPADD*/
nop /*MPADD*/
mov .r19,.r18 /*MPADD*/
add .fpp,2,.r19 /*ADDRLP*/
stg .r18,.r19,0 /*ASGNx*/
L2:
add .fpp,1,.r18 /*ADDRLP*/
ldgn .r18,0,.r19 /*INDIRx*/
gtlo 1,.r20 /*CNSTI*/
add .r20,.r19,.r19
stg .r19,.r18,0 /*ASGNx*/
add .fpp,1,.r18 /*ADDRLP*/
ldgn .r18,0,.r18 /*INDIRx*/
gtlo 1000,.r19 /*CNSTI*/
sub .r19,.r18,.pc /*compare*/
blt L1
mov .fpp,.spp /*free locals*/
```
Listing B.1: The resulting assembly from compiling listing 7.9.